DPLL, First-Order Logic

CS156

Chris Pollett

Oct 22, 2014

Outline

- Introduction
- Horn Clauses and Definite Clauses
- DPLL Algorithm
- Syntax and Semantics of First-order Logic

Introduction

- Over the last couple of weeks, we have been talking about logic-based agents.
- These agents have a knowledge-base that they can tell their percepts to.
- They can also ask their knowledge base questions in order to figure out what to do next.
- We considered formulating knowledge bases using propositional logic and gave an example with the Wumpus World.
- We also gave an example of how to infer facts about the Wumpus World using model checking and theorem proving.
- We said we can determine if A is true in a propositional KB, by taking $(KB \wedge \neg A)$, converting it to CNF, and checking using resolution if this had a refutation.
- We said resolution-based theorem proving was easier because we have only one kind of inference to deal with and to find proofs we can just loop and resolve pairs of existing clauses and see if we get new ones.
- The process stops if either we can derive the empty clause {} or we can't generate any new clauses.
- If the former case we have a refutation, in the latter case we showed how to build a model which satisfies the set of clauses.
- Today, we are going start by looking at a special case of resolution where refutations can be found in linear time.
- We will then look at the DPLL for general clauses.
- Then we are going to look at more complicated logics to formulate KBs with.
- Although we haven't said yet how to use the fact we can infer what's true in a KB to decide what to do next, we did mention, and the book gives an example procedure, of how you can combine steps computing all derivable facts about squares in the Wumpus World so far with an $A^\star$-search. I.e., once a square is proven safe, it made available to the $A^\star$-search to plan a route.
- We will also more about how to decide what to do next after talking about first-order logic when we talk about planning.

Horn Clauses and Definite Clauses

- Before we talk about first-order logic I wanted to mention some last things about resolution.
- So far we have looked at the case where we could have had more than one positive and more than one negative literal.
- In the case where he have exactly one positive literal, we have a **definite clause**. For example, $(L \vee \neg B \vee \neg C)$. This is often written like $L :- B, C$ as an ASCII shorthand for $(B \wedge C) \Rightarrow L$.
- Here is L is the **head** of the clause, and B, C is called the **tail** and the whole clause is called a **rule**.
- Slightly more general than a definite clause is a **Horn clause**, it is allowed to have at most one positive literal. So $(\neg B)$ is a Horn Clause, but not a definite clause. This might be written as $:- B$. If this were a prompt in Prolog, it would be written as $|?- B$ and called a **goal** clause.
- A collection of Horn clauses is called a **Horn Program**, which is a particular kind of **logic program**.
- KBs that are Horn Programs are useful as deciding entailment for these clauses can be done in linear time and space.
- If we want to see if B follows from a KB which is a Horn program we can start with the rules that involve B in the head. For each such rule we could then check each variable in its tail to see if it is a fact in the KB. If not, we look for a rule that involves that variable and try to see if we can derive it and so on. This is called **backward chaining**.
- Alternatively, we can start with the fact in the KB. Then for each rule for which its tail consists of only facts, add its head to the KB. Keep iterating until no change. If we derive B in the process, we know KB entails B this is called **forward-chaining**.

Prolog

- Prolog is a programming language that makes use of the ideas of logic programming.
- It was developed by the group of Alain Colmerauer and Phillipe Rousel in Marseille, France.
- Originally, it was developed to do Natural Language Processing.
- It was used in Japan's AI intensive Fifth Generation Computer Systems project.
- Prolog is a relatively simple -- the first interpreter your professor managed to write was for a stripped down version of Prolog.
- It is sometimes used as scripting language for game AI and for reasoning about web ontologies.
- It was used in Watson -- the AI that defeated Ken Jenning's on the TV show Jeopardy.
- There are several open source Prolog implementations. For example, SWI-Prolog and XSB.

A Prolog Example Logic Program

Consider the Horn program:

```
a. /* lines end with a period, a rule without a tail is called a fact */
b:- a.
c:-b.
d:- a, b.
```

- We could write this program in a file horn_example.pl, say on our Desktop, then load it in at the prolog prompt:

```
?- ['~/Desktop/horn_example.pl'].
```

- At the Prolog prompt, we can now query this knowledge base to see what is true or false.
- If we enter a query at the prompt, Prolog will try to use backtracking to see if the statement is true.
- For example:

```
 ?- a.
true.
 ?- d.
true.
```

How Prolog computes its answers

- To compute the answer, we can imagine Prolog scanning through the rules of the program, looking for the first rule whose head is our query. If this rule if a fact it outputs **true**. This is what happened for the query ?- a. Otherwise, it tries to satisfy each of the conditions in the tail of the rule. In the case of ?- d., we find the rule d:- a, b. and then try to satisfy a. We can because it is a fact. Then we try to satisfy b. To do this we scan the rules from the top and find the rule b:-a. So to satisfy this rule we need a, but we have already derived it is a fact, so we have b. Now that we have a, b we can conclude d, and Prolog outputs true.
- If it was the case that Prolog couldn't satisfy d:- a, b. then Prolog would backtrack and try to find a rule further down in the program with d in the head and try to satisfy that.
- The interpretation of a query on a nonexistent variable like e is an error for SWI-Prolog. We can view this as false: we were not able to derive that variable. This corresponds in some degree to the **closed world assumption (CWA)**, things we don't know about we assume are false. i.e., we should favor minimal models. Favoring maximal models on the other hand corresponds to the **open world assumption (OWA)**.
- A program like:

```
d:-d.
c:-b.
b:-c.
```

followed by a query ?- d or a query ?- b would result in an infinite loop.

- Prolog supports non-Horn rules like:

```
e :- not(a).
f :- false.
```

DPLL Algorithm: A general resolution finding algorithm

- For a general set of starting clauses, resolution might need to do exponentially many resolution steps before deriving the empty clause.
- For example, this is the case for clauses expressing the negation of the pigeonhole principle.
- Nevertheless, we can try to perform optimizations to speed up our basic resolution algorithm.
- One example of such an algorithm is the DPLL algorithm (1962) named after Davis, Putnam, Logemann, and Loveland. It is sometimes called Davis Putnam after an earlier paper 1960.
- DPLL tries to either find a refutation from a set of clauses or find a satisfying assignment. Rather than compute the complete resolution closure before determining an assignment like last day, as we compute the algorithm we will be computing a partial assignment.
- It performs the following three optimizations over the basic resolution algorithm:
  - ***Early Termination****: If in our partial assignment so far, we have made at least one literal in each clause true, then we know the whole statement must be satisfied. So we can stop and output satisfiable given the assignment we have so far. Also, if under the partial assignment, we find some clause in which all the variables are mapped to false, then it stops and outputs false. This case implies a refutation exists.*
  - ***Pure symbol heuristic****: A* ***pure symbol*** *is a symbol that always appears with the same sign in all clauses. For example, in* $\{A, \overline{B}\}$ *,* $\{\overline{B}, \overline{C}\}$ *, and* $\{C, A\}$ *, both* A *and* B *are pure and* C *is impure. Setting a pure symbol's literal so as to make it true can never a clause false. In this case, we can set* A *true and* B *false and not affect the validity of our clauses. This causes the first two clauses to be true so we can remove them from our set of clauses to consider, leaving* $\{C, A\}$ *since* A *has been set to true this is really just the clause* $\{C\}$ *. For this one clause* C *is pure so we can set it true, completing a model.*
  - ***Unit Clause Heuristic****: A* ***unit clause*** *is a clause with just one literal. For DPLL, it can also be clause in which all variables but one have been assigned. There is one truth assignment which can make a unit clause true. So we make this assignment. This might generate more unit clauses. So we then iterate until no more unit clauses derived.*

DPLL Algorithm

```
function DPLL-Satisfiable?(s) returns true or false
   inputs: s a sentence in propositional logic

   clauses := the set of clauses in the CNF representation of s
   symbols := a list of the propositional symbols in s
   return DPLL(clauses, symbols, {})

function DPLL(clauses, symbols, model) returns true or false
   if every clause in clauses is true in model then return true
   if some clause in clauses is false in model then return false
   P, value := FIND-PURE-SYMBOL(symbols, clauses, model)
   if P is non-null then return DPLL(clauses, symbols - P, model ∪ {P=value})
   P, value := FIND-UNIT-CLAUSE(clauses, model)
   if P is non-null then return DPLL(clauses, symbols - P, model ∪ {P=value})
   P := FIRST(symbols); //symbols is a list, this get the first element in the list into P
   rest = REST(symbols); //all the elements in the list excluding the first element
   return DPLL(clauses, rest, model ∪ {P=true}) or
       DPLL(clauses, rest, model ∪ {P=false})
```

First-Order Logic

- Point: Want to be able to reason about sets of objects rather than true/false values.
- Where used: Parsers, Prolog, relational databases, planning
- Syntax

```
Variables: x, y, z, ... -> range over sets.
   Example: x might be an element in a set of colors.
Constants: a, b, c, ...
   Example: Fixed values from a set, 0, "Bob"
Functions: f, g, h, ...
   Example: x + y is a function
Predicates: `P`, `Q`, `R` ...  These take inputs and output true/false.
   Example, x=7 is a predicate.
```

- A **term** is variable or a constant, or built up from these using function symbols and composition.
- An **atomic formula** is a predicate where each of the predicate slots has been filled with a term. For example: IsPrime(X*X+3).
- A **formula** is either an atomic formula or built out of simpler formulas, F_1 and F_2 by one of the following operations.
  1. $\neg(F_1)$
  2. $(F_1 \wedge F_2)$
  3. $(F_1 \vee F_2)$
  4. $(F_1 \Rightarrow F_2)$
  5. $(F_1 \Leftrightarrow F_2)$
  6. $\forall x F_1$ *for some variable x. Reads as for every x, F_1 holds.*
  7. $\exists x F_1$ *for some variable x. Reads as there exists some x, F_1 holds.*

Examples

- Suppose Richard and John are constants, we assume variables range over a space of things.
- We can express facts about two famous kings of England using the predicates, Brother, Married, and King:
  $Brother(Richard, John)$.
- This is an example atomic formula. Let Mother and Father be functions which given a person returns their mother or their father. A more complicated atomic formula expressing that Richard and John are at least half-brothers is:
  $Married(Father(Richard), Mother(John))$
- More complicated non atomic formulas are things like:
  $King(John) \vee King(Richard)$
  $Brother(John, Richard) \wedge Brother(Richard, John)$
  $\forall x(King(x) \Rightarrow Person(x))$