Automating Resolution Refutation

1 Search

Given a set of clauses on which we wish to use resolution, at any stage there are choices:

- Which clauses should be the parents?
  E.g.

  $\neg s \vee r$ $\quad q \vee \neg p$ $\quad p$ $\quad r \vee \neg q$ $\quad s$

  For systematicity, an automated theorem-prover will have a *search rule* that specifies which parents to try next.

- Within parents, which literals should be selected as the complementary pair?
  E.g.

  $p(a) \vee p(b)$ $\quad q(x) \vee \neg p(x)$

  For systematicity, an automated theorem-prover will have a *selection rule* that specifies which literals to try next.

Up to now, we have presented refutation trees, showing the successful resolutions that lie on a derivation of the empty clause from some initial clauses. This is misleading. It makes it look as though a human or a machine could unerringly make the 'right' choices: choose the 'right' parents and the 'right' literals within those parents.

In practice, of course, this isn't so. Proof is a matter of *search*. We obviously want a search strategy which explores as little of the search space as possible (for efficiency) without losing completeness.

This leads us to distinguish three notions:

- The *search space* — *all* possible resolvents.
- The *derivation graph* —the resolvents we actually compute.
- The *refutation tree* —the resolvents that derive $\Box$.

(These three notions are analogous to the three notions we had when we were looking at search previously: the state space, the search tree and the solution path.)

2 Search strategies

(The names I give below for different search strategies are as standard as I can manage, but there seems to be little consistency in the literature.)

Breadth-first resolution (also called *level-saturation resolution*) In this strategy, we compute all resolvents, level by level. This is complete but grossly inefficient.

Linear resolution (also called *ancestry-filtered resolution*) In this strategy, at least one parent is a member of the initial axioms or an ancestor of the other parent. This is also complete and sometimes a little more efficient than breadth-first.

Set of support resolution In this strategy, at least one parent is from the negated query or its descendants. This again is complete (although, strictly, it is only complete if the original clauses are satisfiable) and it even more efficient.

Unit resolution In this strategy, at least one parent must be a unit clause. This is not complete in general, but it can be very efficient.

Input resolution In this strategy, at least one parent is a member of the initial clauses (inc. the negated query). This is not complete in general, but it can be very efficient.

Another way of improving efficiency of reasoning is to move to less expressive logics.

3 Horn clauses

Clausal Form Logic is as expressive as FOPL, but inference on Clausal Form Logic (using resolution refutation) is often more manageable. (It's no better in the worst case, but it is often better in practice.) However, there may still be a lot of search.

As we've seen, the more efficient search strategies (such as unit resolution and input resolution) lose completeness. If efficiency is at a premium, then another option is to restrict expressiveness.

Suppose we allow only a subset of Clausal Form Logic. It will then not be the case that every wff in FOPL when converted to clausal form will end up being in this subset. But, in those cases where our initial axioms and our negated query can be converted to clauses that fall within this subset, then we may be able to use more efficient search strategies.

The resulting system will not be complete in general. But it might be complete with respect to the less expressive logic.

The most common restriction is to limit ourselves to *Horn clauses*.

> Horn clauses are clauses with *at most* one positive literal.

This means that e.g., $\neg p \vee q \vee r$ is now disallowed.

This is arguably a severe loss for a generally intelligent agent, e.g. a robot or softbot that has to operate in a complex environment. It means, for example, that some forms of uncertainty cannot be represented. E.g. we can no longer handle the following clause:

$$colour(clyde, grey) \vee colour(clyde, brown)$$

While this is a severe loss for a generally intelligent agent, it turns out that it may not be such a big loss for many useful 'standalone' agents. (See later in this lecture and subsequent lectures.)

If we restrict ourselves to Horn clauses, resolution refutation becomes complete (with respect to the less expressive logic) for *all* the search strategies we discussed previously — even for the very efficient strategies such as input resolution and unit resolution.

We can get even more efficiency by restricting expressiveness even more. Let's start to do this by drawing the following distinction:

Positive Horn clauses are clauses with *exactly* one positive literal;

Negative Horn clauses are clauses with *no* positive literals.

Some even more efficient search strategies are complete for problems in which

- there is only one negative Horn clause (the query); and
- the initial clauses are all positive Horn clauses.

Again, restricting ourselves in this way is going to be a severe loss if we are trying to build an agent with general intelligence, but may be OK in many 'standalone' agents.

The reason why positive Horn clauses are a useful form of limited expressiveness is because they correspond to *facts* and *rules*.

In the following table, I show some positive Horn clauses. On the left, I write them as we've been writing them so far, using disjunction ($\vee$). On the right, I rewrite them using the conditional operator ($\Rightarrow$):

As disjunctions	As facts & rules
p_1	p_1
$\neg p_1 \vee p_2$	$p_1 \Rightarrow p_2$
$\neg p_1 \vee \neg p_2 \vee p_3$	$(p_1 \wedge p_2) \Rightarrow p_3$
$\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee p_4$	$(p_1 \wedge p_2 \wedge p_3) \Rightarrow p_4$

Much human knowledge can still be encoded even when restricting oneself to positive Horn clauses (facts and rules). Here are examples of various forms of human knowledge that can still be represented:

- *Taxonomic knowledge*, i.e. knowledge of classes, subclasses and instances of those classes:

$$\mathit{elephant}(\mathit{clyde})$$

$$\forall x(\mathit{elephant}(x) \Rightarrow \mathit{mammal}(x))$$

- *Causal knowledge*, e.g. knowledge of medical conditions and their symptoms and other examples of cause and effect:

$$\forall x(\mathit{hasMeasles}(x) \Rightarrow \mathit{hasSpots}(x))$$

$$\forall x(\mathit{standsInRain}(x) \Rightarrow \mathit{getsWet}(x))$$

- *Action knowledge*, e.g. STRIPS operators can be encoded in a logically more 'kosher' fashion in logic using positive Horn clauses:

$$\begin{aligned}&\forall x \forall y \forall z \forall s\\ &\quad((\mathit{on}(x,y,s) \wedge \mathit{clear}(x,s) \wedge \mathit{clear}(z,s)) \Rightarrow\\ &\qquad\qquad \mathit{on}(x,z,\mathit{do}(\mathit{move}(x,y,z),s)))\end{aligned}$$

The fact that this quite limited level of expressiveness is useful is reflected in the fact that it has been adopted as the basis for a number of technologies:

- In Prolog and other logic programming languages, the program consists of a set of facts and rules (positive Horn clauses). To run the program, the user supplies a query, which is in fact a negative Horn clause.

  Even though logic programming languages are based on this quite limited logic, it has been shown that any Turing-machine-computable function can be expressed in this way.
- In deductive databases, the database comprises tables much as in a relational database (and these correspond in logic to atoms, or unit clauses or 'facts'); additionally, the database holds a number of rules, so that new data can be deduced from the database.
- In so-called rule-based systems, causal knowledge is encoded as **if** ... **then** ... rules. These systems have been used to build 'standalone' agents for medical diagnosis, fault-finding, etc. See next lecture.

4 SLD-Resolution

Suppose our initial clauses are only positive Horn clauses (facts and rules) and that our query is a negative Horn clause. There is a very efficient search strategy, made up of several of the other strategies that we have discussed, that we can use to answer this query. We'll call this strategy *SLD-resolution*.

We're going to first describe this strategy using clauses written in their usual form (with disjunctions). But the strategy relies on writing the clauses in a particular way. In positive Horn clauses, the positive literal must be written *leftmost*. For example, the clause $p_2 \vee \neg p_1$ is OK; but writing it as $\neg p_1 \vee p_2$ is not OK.

When choosing parents, we use two search rules (one for each parent):

- Linear/set-of-support: one parent is the query or one of its descendants (hence, it will be a negative Horn clause);
- Input: the other parent is a member of the initial clauses (hence, it will be a positive Horn clause).

And we use a selection rule to decide which literals should form the complementary pair:

- The parents are resolved on their *leftmost* literals only; and
- When you write down the resolvent, the literals in the resolvent will preserve the order from the parent clauses with those from the positive Horn clause preceding those from the negative Horn clause.

In the lecture, we'll show an example of this.

In fact, SLD-resolution can be explained more intuitively if we use the fact and rule notation instead of using our normal clausal form (disjunctions). It also requires that the query, which is a negative Horn clause, be written as a negated conjunction, and that the negation be implicit. For example, if the query is $\neg p_1 \vee \neg p_2$, we know this is the same as $\neg(p_1 \wedge p_2)$, and if we regard the negation as implicitly there, then we would end up writing the query as $p_1 \wedge p_2$.

In the lecture, we'll show an example of SLD-resolution using the fact and rule notation. In the example,

- One parent will be the query or one of its descendants;
- The other parent will be an initial clause (a fact or a rule); and
- These two parents are resolved so that leftmost literal in the query resolves with a fact or the consequent of a rule.

Note from the examples in the lecture that we still have a choice. We can do our SLD-resolution in a breadth-first manner, a depth-first manner or some other manner. In fact, using a depth-first approach is the most common. Prolog, for example, uses depth-first SLD-resolution implemented by a recursive search algorithm. Many rule-based systems also work depth-first (although others make use of other knowledge, such as heuristics, certainity factors, probabilities, etc., to make more informed choices).

Exercise

1. (Not to be submitted for correcting) Look over your answers to the exercises from the previous lecture. Are there any there in which the query was a negative Horn clause and all other clauses were positive Horn clauses? If so, did you use SLD-resolution?

2. (Can be submitted for correcting: past exam question) This question uses the following 'key' for the binary predicate symbols *elem* and *subset*, the binary function symbols *int* and *union*, and the constant symbols *nil*, s_1, s_2 and c:

$\mathit{elem}(x, y)$	:	object x is an element of set y
$\mathit{subset}(x, y)$	:	set x is a subset of set y
$\mathit{int}(x, y)$	:	the intersection of sets x and y
$\mathit{union}(x, y)$	:	the union of sets x and y
nil	:	the empty set
s_1	:	a set called s_1
s_2	:	a set called s_2
c	:	an object called c

(x, y, z and subscripted versions of these will be used as variables.)

(a) Give a natural English paraphrase of the following wff of FOPL:

$$\neg\exists x(\mathit{elem}(x, s_1) \wedge \mathit{elem}(x, s_2))$$

(b) Translate the following sentence of English into FOPL:

"The empty set contains no elements."

(c) Convert the following wff of FOPL into Clausal Form Logic. Show your working.

$$(\exists x\, \mathit{elem}(x, \mathit{int}(s_1, s_2))) \Rightarrow \exists y(\mathit{elem}(y, s_1) \wedge \mathit{elem}(y, s_2))$$

(d) Determine whether the members of the following pairs of atoms unify with each other. If they do, give their *most general unifier* (mgu); if they do not, give a brief explanation.

i. $\mathit{elem}(x, \mathit{int}(y, z))$ and $\mathit{elem}(\mathit{int}(s_1, s_2), x)$

ii. $\mathit{elem}(x, \mathit{int}(x, y))$ and $\mathit{elem}(s_1, z)$

iii. $\mathit{elem}(x, \mathit{int}(x, x))$ and $\mathit{elem}(s_1, \mathit{int}(s_1, \mathit{union}(s_1, s_2)))$

iv. $\mathit{elem}(\mathit{int}(x, y), \mathit{union}(s_1, s_2))$ and $\mathit{elem}(\mathit{int}(s_1, z), z)$

(e) You are given the following *clauses*:

$$\mathit{elem}(f(x_1, y_1), x_1) \vee \mathit{subset}(x_1, y_1)$$
$$\neg\mathit{elem}(f(x_2, y_2), y_2) \vee \mathit{subset}(x_2, y_2)$$

if every element of one set is also an element of a second set,
then the first set is a subset of the second set
(f is a Skolem function)

$$\mathit{elem}(c, s_2)$$

(object c is an element of set s_2)

From these clauses, use *resolution refutation* theorem-proving to show that there is at least one subset of s_2 that contains c, i.e. in FOPL:

$$\exists x(\mathit{subset}(x, s_2) \wedge \mathit{elem}(c, x))$$

Show your working, presenting your proof in the form of a *refutation tree*.

(f) Could one use SLD-resolution to solve part (e)? Explain your answer.