Maggie Johnson
CS109

Resolution

Key Topics

* Conjunctive Normal Form
* Inference Using Resolution
* Resolution in Predicate Logic

This lecture will cover resolution, a very important inference rule. To prepare for doing resolution, we need to become familiar with "conjunctive normal form". This is a method of transforming logical expressions (hypotheses and the conclusion) into a specific form so resolution can work as a rule of inference.

A good place to start is with a precise definition of a logical expression. Here is a neat little recursive definition:

1) Propositional variables and the logical constants T and F are logical expressions.
2) If E and F are logical expressions, then so are:

a) E ^ F	the value of this expression is defined by the AND truth table
b) E v F	the value of this expression is defined by the OR truth table
c) ~E	the value of this expression is defined by the NOT truth table

We need to take a logical expression and transform it to **conjunctive normal form**. This means a logical expression in the form where the variables are OR'd together in groups, and then these groups are ANDed together, as in: (p v q v r) ^ (~q v r). To get a logical expression into this form, we perform the following procedure:

1) Get rid of any operators except AND, OR, and NOT. For example, if we have p -> q as a part of a hypothesis, replace it with ~p v q (you may want to convince yourself that these are equivalent). We can see from our recursive definition of logical expressions that AND, OR and NOT are a complete set of operators, so any logical operator (e.g., NAND, NOT, etc.) can be replaced by expressions involving only AND, OR, NOT.

2) Next, apply DeMorgan's laws to "push" all negations down until they cancel with other negations (as in ~~p = p), or they apply only to propositional variables. Recall DeMorgan's Laws:

DeMorgan's Laws	~(p ^ q) <=> ~p v ~q
	~(p v q) <=> ~p ^ ~q

These laws allow us to successively apply negation inwards on expressions in parenthesis.

3) Finally, apply the distributive law for OR over AND to "push" all OR's below the AND's. "Below" here means the OR is inside the parenthesis while the AND is outside. Here is the distributive law we need: p v (q ^ r) <=> (p v q) ^ (p v r)

Now for an example. We will now use our procedure to transform

p v (q ^ ~(r ^ (s -> t)))

to conjunctive normal form.

1) Replace the (s -> t) by (~s v t) giving p v (q ^ ~(r ^ (~s v t)))
2) Push the first NOT down to a propositional variable using DeMorgan's Laws:

p v (q ^ ~(r ^ (~s v t)))
p v (q ^ (~r v ~(~s v t)))
p v (q ^ (~r v (s ^ ~t)))

3) Use the distributive law for OR over AND to push the first OR below the first AND:

p v (q ^ (~r v (s ^ ~t)))
(p v q) ^ (p v (~r v (s ^ ~t)))

We regroup using the associative law [(p v q) v r <=> p v (q v r)]

(p v q) ^ ((p v ~r) v (s ^ ~t))

Finally use the distributive law again:

(p v q) ^ ((p v ~r v s) ^ (p v ~r v ~t))

And voila, we have an expression where the variables are OR'd together in groups, and then these groups are ANDed together, i.e., conjunctive normal form. Resolution will only work on logical expressions in conjunctive normal form.

Inference Using Resolution

We have mentioned more than once now, that finding proofs is not an easy task. We have even alluded to the fact that any kind of exhaustive search would be inherently exponential. There do exist, however, some useful tricks that help in the exploration one goes through in finding a proof. One useful inference rule called **resolution** is the most basic of these tricks. Resolution is based on the following tautology:

((p v q) ^ (~p v r)) -> (q v r)

Verify that this is a tautology:

The usual way to apply resolution is to convert all the hypotheses into **clauses**, which are OR's of literals. The proof begins with each of these clauses as a line of the proof (since each is a "given"). We then apply the above resolution rule to construct additional lines, which also turn out to be clauses.

Example

Prove the following using resolution: (r -> u), (u -> ~w), (~r -> ~w) |- ~w

The first thing we can do is replace the commas in the hypotheses with AND since that is what the commas mean: (r -> u) ^ (u -> ~w) ^ (~r -> ~w). Now we have to convert this to conjunctive normal form which is trivial in this case. All we have to do is replace the "p -> q" with ~p v q.

(r -> u) ^ (u -> ~w) ^ (~r -> ~w)
(~r v u) ^ (~u v ~w) ^ (r v ~w)

Here is the resolution rule: ((p v q) ^ (~p v r)) -> (q v r). We begin by applying this to the first and third clauses (r plays the role of p).

1)	(~r v u)	
2)	(~u v ~w)	
3)	(r v ~w)	
4)	(u v ~w)	resolution of (1) and (3)
5)	~w	resolution of (2) and (4)

Usually, we do not apply resolution as in the last example. The more common approach is to start with the hypotheses and negate the conclusion. Then, we apply resolution to try and derive a clause with no literals (i.e., it has a value of false). The example above would go like this:

1)	(~r v u)	
2)	(~u v ~w)	
3)	(r v ~w)	
4)	w	Negation of conclusion
5)	(u v ~w)	resolution of (1) and (3)
6)	~w	resolution of (2) and (5)
7)	false	resolution of (4) and (6)

The reason this is valid comes from a law given in the last handout: ~p -> 0 <=> p. Let p be the statement that we want to prove, i.e., (E1, E2, ..., En -> E). Then, ~p is ~(E1, E2, ..., En -> E) or ~(~(E1, E2, ..., En) v E). Using DeMorgan's Laws, we can see that ~p is equivalent to E1 ^ E2 ^ ... ^ En ^ ~E. Thus, to prove p, we can instead prove ~p -> 0 or (E1 ^ E2 ^ ... ^ En ^ ~E) -> 0.

So what's the big deal about resolution? There are couple important points. First, when we combine resolution with negation as above, all of a sudden we have a distinct direction in which to search. We try to prove progressively smaller clauses hoping to get to 0 eventually. That does not mean we will always have success, it just gives us a definite direction. Also, resolution is a **complete** proof procedure for propositional logic. Whenever (E1, E2, ..., En -> E) is a tautology, we can always derive 0 from E1, E2, ..., En and ~E, expressed in clause form. Again, just because the proof exists does not mean it is easy to find.

So you may *still* ask, what's the big deal about resolution? It is of primary importance in AI. One sub-area of AI is knowledge representation, i.e., the problem of finding a language in which to encode knowledge such that a machine can use it. There are several pioneers of AI who believe mathematical logic provides the best knowledge representation language. Whether or not they are correct, logic and algorithms for working with logic, are important. The methods of logic are powerful and they warrant careful study for anyone interested in automatic problem solving.

Resolution is important because it gives a direction to a search. As one AI researcher has put it: "The intended role of knowledge representation in AI is to reduce problems of intelligent action to search problems." In addition, resolution extends nicely from propositional logic to predicate logic. Thus, it provides a very important technique used in automatic theorem-proving and problem-solving, and in the implementation of certain programming languages, e.g., Prolog (more on all of this later).

Resolution in Predicate Logic

As we have mentioned on numerous occasions, predicate logic is much more powerful than propositional logic. In predicate logic, we can manipulate the variables in a way that is just not possible with propositional logic, which gives us a much more expressive language for representing the axioms and theorems of a universe of discourse.

We have also seen that resolution is a rule of inference from propositional logic that is very useful because of the methodical way it directs the search for a proof. The real power of resolution, however, comes when we use it with predicates rather than propositions. To illustrate this, consider the following example.

Example

This example will deal with certain properties of numbers. Here are the given premises: Any prime other than 2 is odd. The square of an odd number is odd. The number 7 is prime. The number 7 is different than 2. We want to prove from *only* these premises that the square of 7 is odd. Of course, we know 7 is odd and we are given that the square of an odd is odd, BUT we are not given the premise that 7 is odd....

We start by representing the premises and conclusion using predicate logic.

- P(x): x is prime
- O(x): x is odd
- E(x,y): x = y
- $s(x) = x^2$

Here are the premises given the above definitions:

- x ((P(x) ^ ~E(x,2)) -> O(x)
- x (O(x) -> O(s(x)))
- P(7)
- ~E(7,2)

The negation of the conclusion (which you may recall is the typical way we use resolution) is ~O(s(7)).

Before we can apply resolution, remember that the premises and the negation of the conclusion must be in conjunctive normal form.

C1: ~P(x) v E(x,2) v O(x)

How did we derive this?

C2: ~O(x) v O(s(x))
C3: P(7)
C4: ~E(7,2)
C5: ~O(s(7))

Note that it is usually much more involved converting predicate expressions to conjunctive normal form. The complications come with all the different combinations of quantifiers one can encounter (e.g., x y z ....). These types of quantified statements require special handling which is beyond the scope of what we can do here.

So here we go with the resolution proof:

C1: ~P(x) v E(x,2) v O(x)	
C2: ~O(x) v O(s(x))	
C3: P(7)	
C4: ~E(7,2)	
C5: ~O(s(7))	
C6: E(7,2) v O(7)	resolution of C1 and C3 (note that we substituted 7 for x in C1; this is called **unification**)
C7: O(7)	resolution of C6 and C4
C8: O(s(7))	resolution of C7 and C2
C9: false	resolution of C8 and C5

Just a quick note on this idea of unification. In applying the resolution rule to predicate clauses, we have to detect situations where a variable in a predicate can be defined (or "unified") given information from other clauses. This is not a trivial procedure since substitutions of a variable in one predicate can ripple through to others. This also is beyond our scope in 109, but refer to the bibliography if you are interested in learning more.

The example above is pretty trivial in content. Let's see if we can't provide a more compelling example.

Example

Sally is studying with Frederico. Frederico is at Tresidder. If any person is studying with another person who is at a particular place, the first person is also at that place. If someone is at a particular place, then he or she can be reached at the phone number for that place.

What is the phone number where Sally can be reached?

Again, this seems trivial, but if one is going to implement a problem-solving algorithm, we can't skip over the obvious steps; we have to define everything very precisely.

First, get everything in predicate logic form:

- SW(x, y): x is studying with y
- A(x,y): x is at place y
- R(x,y): x can be reached at phone number y
- ph(x): phone number of place x

Here are the premises:

- SW(Sally, Frederico)

- A(Frederico, Tresidder)
- x y z (SW(x,y) ^ A(y,z) -> A(x,z))
- x y (A(x, y) -> R(x, ph(y)))

Here is the negation of the conclusion: ~ xR(Sally, x)

With some applications of the implication and DeMorgan's Laws we get the following clauses. Note we have also modified the variables to avoid confusion between predicates. This is a part of the conversion and unification algorithms mentioned above.

C1: SW(Sally, Frederico)
C2: A(Frederico, Tresidder)
C3: ~SW(x,y) v ~A(y,z) v A(x,z)
C4: ~A(u, v) v R(u, ph(v))
C5: ~R(Sally, w)

The resolution steps that produce the null clause are:

C6: ~A(Sally, v)	resolution and unification of C4 and C5
C7: ~SW(Sally, y) v ~A(y, v)	resolution and unification of C3 and C6
C8: ~SW(Sally, Frederico)	resolution of C2 and C7
C9: false	resolution of C1 and C8

You will notice that the problem asked us to define the phone number where Sally can be reached. How can we modify the premises to get the result that we need?

Logic Programming and Prolog

A theorem-proving program takes as input a set of axioms and some formula to be proven. The output consists of information about whether or not a proof was found, and if so, what unifications were done to find it. If we consider the input axioms to be a type of program and the theorem prover a kind of interpreter, then we can "program in logic". By adding some extra features to help guide the interpreter in the proof, and some printing functions, we have ourselves a programming language. This is exactly what Prolog is.

Logic programming is commonly done using predicate logic clauses called **Horn clauses**. Horn clauses are clauses that satisfy a particular restriction: at most one of the literals in each clause must be unnegated. For example:

~P v Q

~P1 v ~P2 v ... v ~Pk v Q

The reason for this is we can then rewrite them back to implication or “goal-oriented” form:

P -> Q
P1 ^ P2 ^ ... ^Pk -> Q

or in goal-oriented form:

Q <- P
Q <- P1 ^ P2 ^ ... ^Pk

Horn clauses in goal-oriented form are used to program in Prolog. A simple program using resolution on Horn clauses in Prolog is shown below. The premises stated in English are:

1) X is a grandson of Y, if for some Z, X is a son of Z and Y is a parent of Z.
2) Huey is a son of Dewey
3) Louie is a parent of Dewey.

Who is the grandson of Louie? Notice we use the Prolog symbol “:-” instead of “<-” of Horn clauses:

```
grandson(X,Y) :- son(X,Z), parent(Y,Z).
son(Huey, Dewey).
parent(Louie, Dewey).

?- grandson(W, Louie).
```

Prolog proceeds to answer this question by performing resolution and unification. It searches through the database of clauses looking for unifiable terms and resolvable clauses. Eventually, Huey will be substituted for W solving the above problem. This example is very simple. With a database of hundreds to thousands of clauses and rules, the search may take a long time resulting in many dead-ends. Prolog interpreters implement a technique called **backtracking** to recover from dead-end searches and thus, continue the search down a different path.

Here is another example, in case you are going out to dinner tonight:

```
redwine(beaujolais).
redwine(burgundy).
redwine(merlot).

whitewine(chardonnay).
```

```
whitewine(riesling).

meat(steak).
meat(lamb).
fish(salmon).

goodwine(Wine) :- maincourse(Entree), meat(Entree), redwine(Wine).
goodwine(Wine) :- maincourse(Entree), fish(Entree), whitewine(Wine).

maincourse(salmon).

?- goodwine(X).
```

Bibliography

Resolution was first defined and used in:

J.A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the Association for Computing Machinery*, 12(1), 1965.

The following textbook provides more material on conversion of predicate logic expressions to conjunctive normal form and unification algorithms. Actually, any AI textbook will have this information.

P.H. Winston, *Artificial Intelligence (Second Edition)*, Reading, MA: Addison-Wesley, 1984.

For more on Prolog:

W. Clocksin, C. Mellish, *Programming in Prolog*, New York: Springer-Verlag, 1981.

I. Bratko, *Prolog Programming for Artificial Intelligence*, Reading, MA: Addison-Wesley, 1986.

For more on logic programming in general:

R.A. Kowalski, *Predicate Logic as a Programming Language*, Amsterdam: North-Holland, 1977.

Historical Notes

During the sixties, there was a great interest in automatic theorem-proving. A primary researcher in this area was Robert Kowalski at the University of Edinburgh, who

concentrated on logic programming, i.e., the use of computers to make controlled logical inferences.

The idea of using a theorem prover as a program interpreter was first tried in the PLANNER programming language (which was embedded in LISP) developed by C. Hewitt in his Ph.D. dissertation at MIT in 1971. Incorporating a predicate logic style into a programming language was first achieved in an accepted way in Prolog which was developed by Alain Colmerauer, Philippe Roussel and a group of researchers at the University of Marseille in the early 1970's.