

**Artificial Intelligence:
Complexity of Resolution Refutation
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module – 07
Lecture - 05**

Okay so we have been looking at the resolution refutation method. And we have found a method that is sound and complete and semi decidable that's the best you can do in first order logic. But lets look at the complexity side of it. So how long does it take to find a proof basically, sometimes what is even the length of the proof. Now it turns out that in 1985 one guy can Haken he showed that he showed for propositional logic. One thing about propositional logic is that it is decidable as opposed to first order logic which we have seen is semi decidable. All the methods that we have seen forward chaining backward chaining resolution can run into infinite loop. But in propositional logic since you are always working with a finite set of clauses of finite set of formulae, eventually you will exhaust all possibilities and you will say okay I am not able to prove it essentially. In first order logic we cannot say that we are not able to prove it. For example we try to show that $0 < 0$? And we were not able to show that given the knowledge base we had and it kept asking question whether $1 < 0$, whether $2 < 0$, whether $3 < 0$ and it could go on infinitely along with sequence.

In propositional logic one can terminate, so we can say no other possibilities exist. But what Haken told was that given a set of clauses that we could produce a set of n clauses the shortest proof is exponential in length. Which means that the number of derivation steps that you do is exponential in n essentially. So I should really say in n . so if the proof itself is exponentially long then the proof finding process because remember we are doing some kind of search, we may do useless inferences, we may do some useless application of the resolution rule and so on before we come up with the final proof. If the proof itself is exponentially long then the proof finding process will be more difficult essentially.


So the first thing we understand is that the task of finding a proof is not going to be easy in first order logic. Because even in propositional case there are formulae for which the proof is exponentially long and sometime I had mentioned earlier that if we take a formula in general first order logic and you covert it into clause form the length of formula will increase, so the number of clauses can go up. You must also keep in mind the very well known result by Stephen Cook who actually earlier had shown and he define the notion of NP Completeness and infact the whole concept of NP completeness is centered around the SAT problem and he showed that the SAT problem is NP complete.

Now remember that the SAT problem is a propositional logic problem where you are given a formula in clause form or in some form actually. But generally when we talk about SAT we say 2-SAT, 3-SAT so anything which is more complex than 3-SAT is in the NP complete set which means that it's a hard problem. So in practice if you are trying to show that something is satisfiable, or you are trying to show that something is unsatisfiable which is kind of the other side, you can run into exponential time . which is why very often we use randomized methods. So this is just to highlight the fact that it can be hard trying to find the proof in first order logic.

Now there is one approach which is due to a mathematician called Herbrand which is called Herbrand's theorem. So before I state the theorems let you give a couple of definitions. We have something called he Herbrand universe, you are given certain sentences those set of sentences will refer to certain objects, to some constants in the domain. So lets say the constants in the domain are a and b just for illustration process. The herbrand universe is the names of objects that you can construct using the constants that are available in the given set of sentences and the functions that are available. So if the constants given to you are a and b then the herbrand universe is a and b, lets say f is a function given to you f of a and f of f of a, and so on , may be g of a, g is another function given to you, g of f of a, all kinds of things, whatever is possible, you construct.

(Refer Slide Time: 7:18)

Resolution Refutation - complexity
 Armin Haken (1985) - Propositional logic
 Given $\{C_1, \dots, C_m\}$
 the shortest proof is exponential in m .
 Stephen Cook (1972) - SAT - NP-complete
 Herbrand's Theorem. \downarrow Randomized methods.
 Herbrand Universe = $\{a, b, f(a), f(f(a)), \dots, g(a), g(f(a)) \dots\}$

 NPTEL

16 / 16

Then the Herbrand base is instances with elements of the herbrand universe substituted for variables. So the herbrand base is a set of sentences that you

arrive at by substituting for variables all the constants that we can invent. Okay what are the constants that we can invent? So those are the constants that are given to you to start with a , b and then by applying the function symbols, f of a , f of b and so on. If you substitute those elements of herbrand universe into the formulae then you essentially come up with a set of sentence which are without variables in other words which are propositional in nature. So ofcourse the first hope was that can we now treat this as a propositional logic problem but obviously there are no free lunches in computing anywhere.

You can see that the moment you have a function symbol and the language is unrestricted in the sense that a function symbol can apply to any term then you will have an infinite number of terms. So you started about with lets say we are talking about a set of students and there are lets say 15 students so we started out with 15 constants and then some function definition lets say fatherof or whatever function you can think of or bestfriendof. Then you could define more terms. The whole idea of first order logic was to not go around making infinite statements but unfortunately if you have function symbols and if you keep applying them in an unrestricted manner even to a small set of constants you will end up generating an infinite herbrand universe. In which case the propositional version of the language which is the herbrand base will also be infinite. So we will not get the benefit of working with the finite propositional set of sentences for which there is termination.

So what herbrand theorem says is that if H is the herbrand base of a language or a set of sentences S then if H is unsatisfiable then so is S . in other words it suffices to show that if you have been working with a herbrand base if you can show that a set of sentences is unsatisfiable then you have shown that the original set of sentences is also unsatisfiable. Now obviously people have thought of various ways we will not get into those here, for example you could work with a type kind of language in which you cannot apply functions indiscriminately, for example a function successorOf you could apply only to numbers. The function fatherof you can apply only to humans and so on and so forth and then you could try to restrict the set of herbrand universe. Or if you are happy or lucky enough to work with a set of sentences in which there are no function symbols then you can convert everything into propositional case and try to solve it. So that's an approach to those of you who have studied planning. It has been taken by the planning community wherethey say instead of having all operators which are variables you can convert them into ground operators. So you come with a propositional case and try to solve it.

But that doesn't work so we also look at search heuristics. I will just mention a couple of examples here and then we will move to looking at a subset of FOL which is more tractable. Remember that our resolution method says that choose two clauses, resolve them, add resolvent to the set and you keep doing this. Just take two clauses, resolve them, add resolvent to the clauses and so on and so forth. The question is which clauses? Because everything depends on that. If you happen to choose the right clauses then you will come up with derivation quickly.

So there are couple strategies that people have talked about. So one thing you should observe is that if I take two clauses, let's say C_1 length is equal to 3 and then there is length C_2 is equal to 4. Then the resolvent could be as much as 5. So you know will remove one disjunct from C_1 and one from C_2 because they will cancel. The remaining you have to keep. So the 2 plus 3 which is 5.

(Refer Slide Time: 14:21)

Search Heuristics

Choose 2 clauses from the set
 Resolve them
 Add resolvent to the set

Which clause?

length $|C_1| = 3$, length $|C_2| = 4$, Resolvent could be of length 5!

NPTEL

And what do we want we want the null clause to be 0, no elements in the null clause. You want to work towards the null clause so it would not seem like a good idea to work on two clauses which are long and produce a resolvent which is long, doesn't solve the purpose. So one strategy is called Unit Preference. And as long as possible choose one of the clauses to be of length 1. Then you can see that if you choose one clause of length n and the other clause of length 1 then the resolvent will be of length n minus 1 because one of them will get cancelled. So then you will end up with smaller and smaller clauses. So that's the unit preference strategy that as long as possible choose one of the clauses to be a unit clause. If you happen to choose both a unit clauses it means that you have come to the end of procedure.

Another strategy is called set of support. This says that one of the clauses must be a descendent of the goal. Remember that the goal is negated. What is the intuition behind this? The intuition behind this is the reason why our set of formulas is unsatisfiable is because we negated the goal and added to the set and then it became unsatisfied. So whatever contradictions that we want to derive must come from the negated rules. So the set of support strategy says that one of the clauses must be a descendent of the goal clause or the goal clause was one of the

ancestors of this clause. In fact we will see an example, an inset of this strategy in a short while.

Another approach is to work with a subset of language. And this subset is called Horn Clause subset. That you don't work with the full language but you don't allow certain things to decide. And as we will see what you don't allow to decide is disjunction. You can't make statements which kind of uncertainty about it. You cannot say that colour A is green or colour B is green because then it's going to lead to complexity. What are horn clauses? Horn clauses have at most one positive literal which means the horn clause has a structure of NOT P, its to be in first order case but I am writing some symbols here OR NOT Q or NOT R . or you could have something like R, you are allowed to have. Or you could also have something like not S or not T. now these two clauses are called positive or also called definite. And such a clause is called a negative clause.

(Refer Slide Time: 19:12)

The screenshot shows a digital whiteboard with the following handwritten text:

- Horn Clause Subset of FOL
- ↳ at most one positive literal
- ex: $\neg P \vee \neg Q \vee R$
- R
- $\neg S \vee \neg T$ → negative clause.
- positive / definite clauses (indicated by a bracket next to the first two examples)

The whiteboard also features an NPTEL logo in the bottom left corner and a slide number '18 / 10' in the bottom right corner.

So in the horn clause subset of the language you are allowed in every clause, remember it's in clause form which is CNF form, every clause can have at most one positive literal. So it can have 0 but it cannot have more than 1. So things like Green a or Green b are not allowed. Because this has got two positive literals. And you can also get the feel that these are the kind of statements which led to incompleteness of forward chaining and backward chaining. If you start talking about disjunction then that's the problem that can happen.

Now let's revisit the resolution rule with horn clauses. So let's construct a small table where the input clause can be positive or negative. So there are two clauses we are

selecting. So let's take two positive clauses and we are talking about horn clauses right. What will be the result of resolving two positive clauses? So let me just take an example so P or not R and R or not P , let me use something else, Q . so these are two positive clauses, this is positive and this is positive. You can see that there are only two positive literals in these two clauses, both are positive so one must have one positive literal and other must have other positive. One of them will get cancelled because that's what the resolution method says. So in our example this will get cancelled with this and what we will get is P or not Q . and the result is positive.

(Refer Slide Time: 22:05)

Horn Clause subset of FOL
 ↳ at most one positive literal
 ex: $\neg P \vee \neg Q \vee R$
 R
 $\neg S \vee \neg T$ → negative clause.

~~$\neg \text{huan}(a) \vee \neg \text{huan}(b)$~~

positive / definite clauses

Resolution Rule with Horn Clauses

		+	-
+	$P \vee R$ $R \vee Q$		
-	$P \vee \neg Q$		

NPTEL

So when we resolve a positive clause with a positive clause, we will get as a result a positive clause. If we resolve a positive clause with a negative clause, for example if we take a positive clause, let's take the same one, P or not R , and let's take a negative clause let's say it must contain not P because we want to be able to resolve something and it has something else not Q or not S or something like that. So we will end up removing the single positive literal which is there and we will end up with a formula which is not R or not Q or not S . so if we resolve a positive with a negative clause then we will get a negative clause as output. And by symmetry the same if we resolve a negative clause with a positive clause.

And we are talking about horn clauses. See non horn clauses may have more than one positive clause which means that the positive clause, if you have two positive literals and you resolve it with a negative clause you would still be left with one. So you could end up with positive results. But if you are working with horn clauses then this is the table that we are constructing which says that if you resolve two positive clauses you will get a positive clause. If you resolve a positive clause with a

negative clause you will get a negative clause. What about the case of resolving a negative clause with a negative clause. Its not possible because you need a possible literal so this case doesn't exist at all.

(Refer Slide Time: 23:59)

Horn Clause subset of FOL
 ↳ at most one positive literal
 ex: $\neg P \vee \neg Q \vee R$ } positive / definite clauses
 R
 $\neg S \vee \neg T$ } negative clause.

~~$\neg \text{huan}(a) \vee \neg \text{huan}(b)$~~

Resolution Rule with Horn Clauses

	+	-
+	+	-
-	-	X

Diagram showing resolution of $P \vee R$ and $\neg R \vee Q$ to $P \vee Q$.
 Diagram showing resolution of $P \vee R$ and $\neg P \vee \neg Q \vee \neg S$ to $\neg R \vee \neg Q \vee \neg S$.

NPTEL

Okay now the next thing we want to show is for horn clauses and if goal is negative then there always exists a derivation in which every resolvent is negative. This is something that somebody has proved but we will not try to prove it. Instead we will try to look at an example and sort of justify this statement. So let me just work with an example. Lets say we have three negative clauses sorry three horn clauses. And let us say I have a derivation in which I resolve this with this. So this C will cancel with not C. we will be left with not E or not F or D. and then if I resolve this with this, I get not E or not F or not A. so I have a negative goal but in my derivation I have a positive clause. And what one can kind of demonstrate is that we can transform every such derivation into one which doesn't have a negative, doesn't have a positive resolvent. So the basic idea is this, look for in some sense the lowest positive resolvent, lowest meaning which occurs latest in the derivation and remove that.

So how can we remove this particular positive resolvent? We can do it by doing the following derivation instead.

(Refer Slide Time: 27:10)

← Movies & TV

NPTEL

Horn Clauses — if goal is negative,
then there always exists a derivation
in which every resolvent is negative

Ex

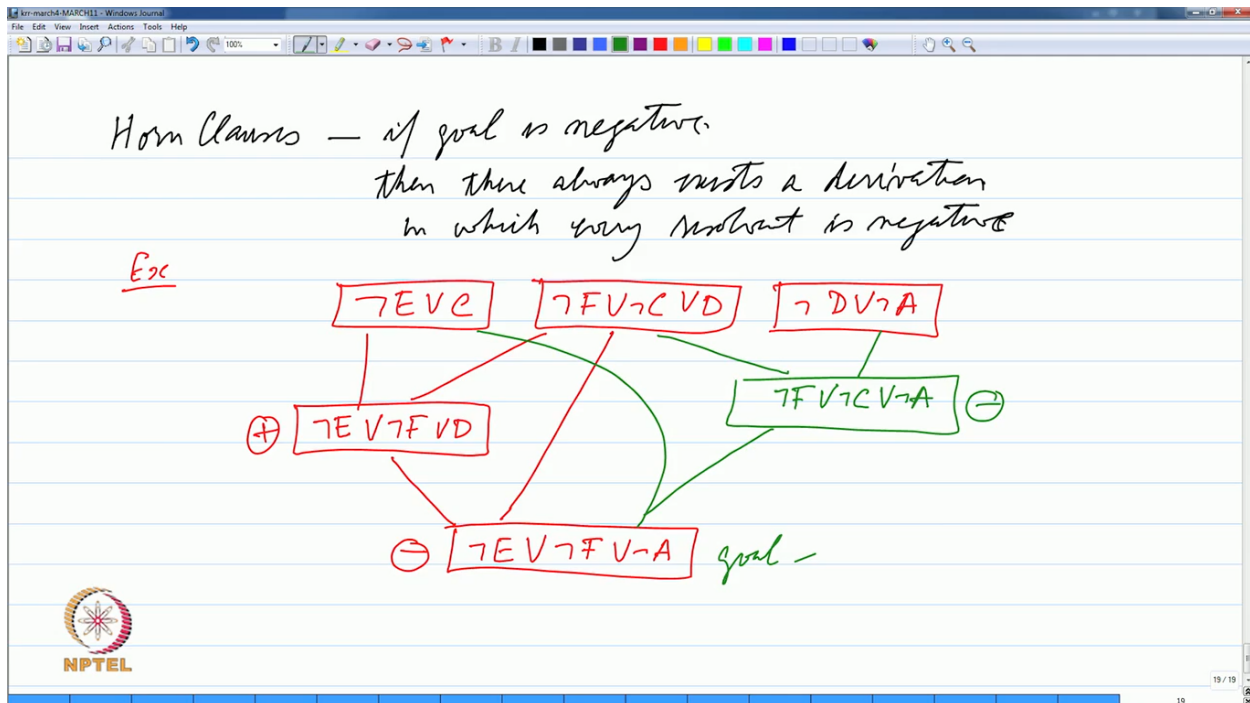
$\neg EVC$ $\neg FV \neg CVD$ $\neg DV \neg A$
 $\oplus \neg EV \neg FVD$
 $\ominus \neg EV \neg FV \neg A$

19/19

2:00 PM 10/13/2016

That instead of resolving that with this we will resolve this with this. So here D will cancel with not D so we will get not F or not C or not A which is a negative clause. And then we will resolve this with this. So not C will cancel with C so we will be left with same clause. So you can see that this is the goal clause that we are talking about, goal is negative. We have two possible derivations, one which has a positive resolvent along the way and one which doesn't have.

(Refer Slide Time: 28:13)



So you can just take it for granted or you can just try to work out an algorithm to do this. Pick the lowest, lowest meaning the latest positive resolvent. And by doing something like this because there will be three parents involved that's why I have drawn the subset here. If you just redo the derivations amongst those three parents instead of deriving this positive clause on the left you can derive the negative clause which is shown on the right and you will end up with the same negative goal that you had to derive. So you can always have a system derivation in which there are only negative clauses.

Now we look at again we are focusing on horn clauses and we will look at something called SLD derivation. Just for the sake of this this stands for selected literals, this stand for linear pattern and D stands for definite clauses. So remember definite is just another term for positive clauses. Now to jump ahead a little bit when we talk about positive clauses if you think about it a little bit you are talking about Prolog program. Because what is a positive clause? If you look at a positive clause like not P or not Q or not R or S. this is just another way of writing. P or Q or R implies S. which in prolog you would have written like this. So a prolog program is a set of definite clauses or positive clauses.

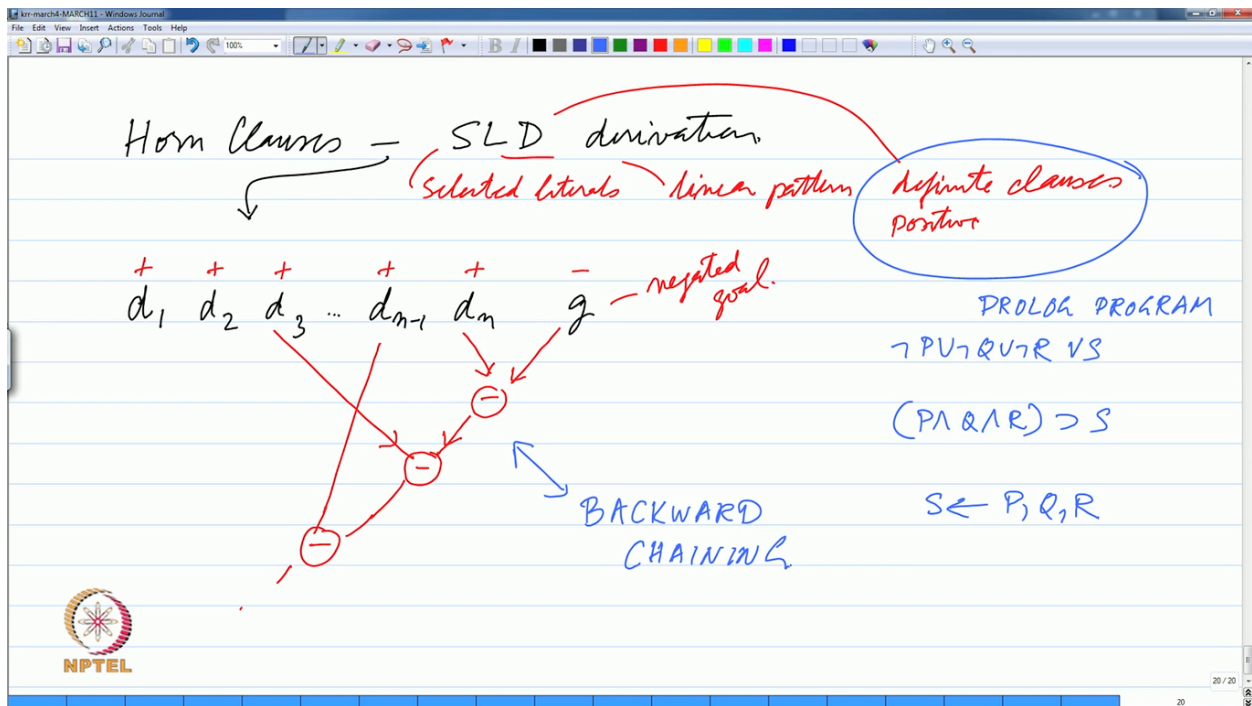
And this SLD resolution is a special derivation which says that which has the following structure and then I will talk about it. So lets say we have a set of clauses $d_1 d_2 d_3 \dots d_{n-1} d_n$ and so let us say all these are positives so you can think of it as prolog program and this the negative which you can think of as negative goal. So remember the resolution refutation method you take a goal negate is and add it to the set of clauses and then you do the resolution. Now a SLD derivation is a derivation which has the structure that every resolvent is a negative clause and we

have shown that that will be the case. We have not shown we have sort of illustrated that that can be the case.

And the parents the every resolvent is the last clause that was derived the last resolvent. And one of the definite clauses. So which if we draw a diagram will look like this. So I take the negated goal and resolve it with one of the clauses. So remember that positive and negative will give us a negative clause. But now in SLD derivation I must use this as one of the input clauses. So one of my two clauses will always be the last clause I have generated, the last resolvent I have generated. So I can only take a child of this. So I can for example take something from here. So observe that I have to take something from the set of definite clauses because I need a positive and negative clause to be able to resolve. I cannot anyway go back to one of the negative parents.

So this will be another negative clause and I have to use this and may be this one I will get another negative clause and so on. So I will ask you to look at one of the program that we have written, Plus or something like that and observe that this whole process is backward chaining.

(Refer Slide Time: 34:15)



Because we start with the goal then this becomes the subgoal so if you identify negative clauses with goals then you start with the negative goal then you come with the subgoal then next subgoal and then to the next subgoal till the process terminates. So you can actually think of this as a prolog program with backward chaining. What is the interesting thing that we know? Which makes this interesting is that this property holds that, remember we are talking about horn clauses, that if we can derive a null clause from a set of clauses and we are interested in the

resolution method because we have shown that it is complete for deriving the null clause. Then so this is for horn clauses. And if this SLD derivation will be denoted by this.

(Refer Slide Time: 35:59)

Horn Clauses - SLD derivation

selected literals linear pattern

definite clauses positive

$d_1^+ d_2^+ d_3^+ \dots d_{m-1}^+ d_m^+$ g^- - negated goal.

subgoal

BACKWARD CHAINING

PROLOG PROGRAM

$\lceil PV \rceil QV \lceil R VS$

$(P1 \& AR) > S$

$S \leftarrow P, Q, R$

NPTEL

For Horn Clauses $S \vdash []$ IFF $S \vdash_{SLD} []$

This statement is what is interesting. It says that and we have tried to hint towards this by saying that we can always transform a derivation in which there a positive resolvents into one which has only negative resolvents. And then we are making a claim that if we have this kind of only positive definite clauses which is like a prolog program and one negative clause which comes from the goal then process is like backward chaining. So we studied prolog as backward chaining but we should now be able to identify backward chaining with SLD derivation which horn clauses. And what other people showed was that this thing holds that if you can derive a null clause using any kind of derivation in horn clause then you can derive the null clause using SLD derivation using horn clause.

The nice thing about SLD derivation is that this is a linear process. You don't have too many choices. Computational complexity comes out of choices, which clauses to take. In SLD derivation this will be frozen for you. You can only make last resolvent as one of the parents to resolve and the other parent will automatically, well you have choice as to which definite clauses to pick but you can only pick that which resolves with one of the negative literals in the parent. So if my parent looks like, lets say not R or not S, some negative clause. Remember that we are always generating negative clauses then I can pick only that positive clause which either contain R or another positive clause which contains S. I cannot randomly pick other

kind of clauses. So first of all I am forced through a linear process that I must pick as one parent the last resolvent that was generated. The other parent choice is almost automatic, almost automatic in the sense that with this parent not R not S there are two separate clauses which you can pick but the choice is limited to removing some of those negative clauses.

The price that we have to pay for this ease of complexity is that the onus is now as we have seen when we were looking at backward chaining is on the user to write rules in a particular fashion. We have said that prolog will always choose the first rule that matches in the order in which you write. So I will leave you with this. I will ask you to do a little bit of reading about this. With this class today we have finished the first seven chapters of Reckman and Lewis. Not necessarily in the order they are given in the book and we have also finished chapters 12 and 13 of my book which are the two references we are using. So we have kind of dealt with general theorem proving in some sense. In the next class we will come back to the presentation issues and we will try to search for clauses to make inferences and so on why not put together relevant piece of information together in one place in some sense and we will look at frames which was introduced by Minsky which was the basis of what we now call the object oriented programming. So we will have a change of flavor from the next class onwards.