

Artificial Intelligence:
Data Retrieval in Backward Chaining
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module - 06
Lecture - 01

Okay so we return to this topic of deductive retrieval. And we want to look at an alternative approach which is backward chaining. So let's first recall what we have studied so far. So far we have looked at forward chaining. And what forward chaining did was that given a knowledge base which contains some facts. So let's say these small circles represent facts and some rules whatever inferences we can make we go ahead and make. So if we can somehow add a new sentence to the knowledge base you add it. Add another new sentence so you keep adding new sentence so obviously you can see that this is kind of a uncontrolled process and the hope is that if you want to prove a sentence α let's say it is here then eventually you will end up adding it.

Now you can see that the number of inferences you can make is proportional in some sense to the size of the knowledge base. So the more the facts and rules you have the more the new facts and rules you can add. So it kind of tends to go rapidly and you are only hoping that at some point you will end up adding the fact α to that. So you can say forward chaining is eager as some people call it Assertion Time. And by this we mean that if you can make some inference go ahead and make it. Whenever you assert a fact and if that fact leads to a new fact infer the facts. Ofcourse you may be constrained by the fact that you can do only one inference at a time so you may have to choose a particular strategy and we saw that when we looked at OPS5 and Rete net that it allowed us to use some sort of a conflict resolution strategy which decided which rule to apply to which fact.

But that's a different matter. But you make whatever inferences you can make and the hope is that eventually you will make the inference you are interested in which is the sentence α that we want to generate. So this is also data driven. Now what we want to look at is this process of backward chaining which is in contrast lazy. So the evaluation is lazy which means its query time. Whenever you want a certain formula you want to query about a certain formula then you make start doing the inferences. Its also as opposed to data driven its also goal directed. That another word that we often use is teleological. Goal oriented trying to achieve a certain goal. So if you remember what forward chaining the basic idea behind forward chaining was that we use modified modus ponens rules which said that if you have let's say β implies γ and you are given β prime and if you can unify them with some substitution θ then from this infer $\gamma \theta$. Apply that substitution to γ and you get θ so that's data driven.

So you are moving from the antecedents of the rule towards the consequent. In backward chaining what we do is so if you want to say that we have a query lets say in this case this query is alpha that we are interested in. and we will write this by saying Show alpha. Or we can write it by saying Goal alpha. So if you have a goal alpha which means you want to show that alpha is true. Then we say that if we have a rule lets say again of the same kind beta implies gamma and if you can unify alpha and gamma again lets say with theta. Then from this you say show beta theta or you can say that goal is beta theta. So in some sense this was the query or goal and this is the subquery or subgoal.

(Refer Slide Time: 7:19)

Deductive Retrieval with Backward Chaining

Forward Chaining
 EAGER - Assertion Time Inference
 Data driven

Backward Chaining
 LAZY - Query Time Inference.
 Goal directed.
 Teleological

KB (Knowledge Base) contains facts. A goal $\beta \supset \gamma$ is derived. This goal is unified with a rule $\beta \rightarrow \gamma$ using a substitution θ . The result is a subgoal $\beta \theta$ and a subquery 'Show $\beta \theta$ '.

NPTEL

So let me illustrate this with our favorite example which said that Man x implies Mortal x. and lets say we have certain facts in our knowledge base as we have been talking Man Socrates. Then our goal is Mortal Socrates then you can see that we can unify this with this by saying x is equal to Socrates. And we can reduce it to a subgoal Man Socrates. And this subgoal we obtain by applying this substitution x is equal to Socrates to the left side of the antecedent. And so we have a goal Man Socrates. If you can find a goal sentence in the knowledge base we stop. So that is a termination criteria. So in backward chaining we are moving from the consequent to the antecedent. Now interesting this is that we can extend this idea to existential queries. So we can say that our goal is let me first write in first order logic. There exists an x Mortal x. this is the kind of queries that you make typically with databases. Is there an employee who has been working for three years and whose salary is more than 10 thousand rupees. Then you want to retrieve answers to that. That's why we call it deductive retrieval in this case.

Now the thing about our existential query is that we will reverse the skolemization convention. Our skolemization convention so far was that universally quantified variable will be marked with a question mark in front of the variable name. but for goals we will reverse this convention and it still stands for existential. Which means that we will express this query as $\text{Mortal } x$. so remember that now this question mark x stands for an existential query. So we are not saying that show that everybody is mortal. We are saying that is there someone who is mortal. Is there some x which is mortal.

Now we can see that with this query and with this same fact we can now have a subquery or subgoal. So it happens that I have used same variable name but ideally I should have used a different variable name. and then you would have said for example y is equal to x or something like that. But it doesn't matter as long as you remember that that should be done. So we have a subquery called $\text{man } x$. so again we look into our database and we see that yes there is a sentence of the kind man Socrates and we can answer yes to this. So in the first case we can return an answer saying yes. In the second case with an existential query we can return an answer yes again that yes the sentence is true that there exists an x such that x is mortal. But we can also say that x is equal to Socrates. We can return a value essentially. So instead of Man Socrates if we had another sentence which said that $\text{Man father lets say Suresh}$ then if we had to match this with this then our answer would be yes father of Suresh is mortal. Remember the original query was is there someone who is mortal we could have answered yes Suresh's father is mortal or Socrates is mortal. And things like that so we can work with existential queries and return answers to that.

(Refer Slide Time: 13:42)

Example

$\text{Man} (?x) \supset \text{Mortal} (?x)$

$\text{Man} (\text{Socrates})$

$\text{Man} (\text{father} (\text{suresh}))$

Yes
father (suresh) is Mortal

Sub goal

goal : $\text{Mortal} (\text{Socrates})$

$x = \text{Socrates}$

goal : $\text{Man} (\text{Socrates}) \rightarrow \text{YES}$

EXISTENTIAL QUERIES

goal : $\exists x \text{Mortal}(x)$

Reverse Skolemization

?x - Existential Convention

$\text{Mortal} (?x) \rightarrow \text{YES}, x = \text{Socrates}$

NPTEL

2/2

The interesting thing is we can now extend this ability to do something which we call as programming. So let's take an example. Append two lists you are given two lists and you need to write a program to append those two lists. So for example if you are given the list. So we will use square brackets to represent list. This is a list. You must return an appended list is This is a list. So this is a task you may have written a program to append two lists there are various ways of doing that. What we do with logic is that we are saying ok what we do when we are working with logic is that we give a logical description of what does it mean to append. And we do that by defining sentences. So how do we do that. So in the case of append we say that this sentence is true append let me call it x1. So our schema is append list1 list2 list3 where this is list3 this is list2 this is list1. So our predicate append is true if the first and the second give you a third list. So if you say append a b c d abcd then it is true. If I say something like append bc ad then this is false.

So that's the meaning of the predicate append. That it will be true if the first argument appended to the second argument gives you the third argument as a list and all the three arguments are lists. And this one statement what we have written about append says that if you append the empty list with any list which you call x1 then you get x1. Which is obviously a true statement because you append an empty list to a list you get the first list back. The second sentence we write is that if you were to append x y to give you z then okay I need to reduce the little bit of notation here. We will represent a list by cons pair notation. So there is this notation called as cons pair notation which is how internally lists are represented and this cons word comes from the language Lisp so you have studied the language lisp and you know that there is an operator called cons which takes two arguments. Cons head and tail so a list will always be represented as cons head with tail. So if I have a list made of b and c then it would be represented as an element b here followed by element c here followed by nil. So the list bc would be represented as cons of b with cons of c with nil. So cons is a pair it tells you which is a head and which is a tail.

(Refer Slide Time: 19:41)

PROGRAMMING

Example - append two lists eg: $[this\ is]$ $[a\ list]$

\uparrow \downarrow
 logical description [this is a list] - list3
} define append.

SCHEMA

append(list1, list2, list3)

1. append(nil, ?x1, ?x1)

2. append(?x, ?y, ?z) \Rightarrow

append([ab][cd][abcd]) true
 append([bc][ad][abcd]) false
 CONS PAIR notation
 cons(head, tail)
 cons(b, cons(c, nil))

NPTEL

And in this tree you see here there are two cons pair each has a left hand side and a right hand side. The left hand side points to the head of the list the right hand side points to the tail of the list. And for the second cons pair head and c and the tail is empty list. So the list containing bc is stored as cons b cons c nil essentially. So with that definition we can now write this append which says that if you can append x to y to give you z then you can append something preceding x so cons something lets call it a x with the same y to give you what you originally got preceded by the same a essentially. So cons a z. this is obviously a true formula because what I am saying here is for example if I have this one append lets say 1 comma 2 comma 5 comma 6 gives me 1 comma 2 comma 5 comma 6 this means that if I were to put anything before 1 in the first list. So lets say 8 then this would also be true. 8 comma 1 comma 2 5 comma 6 8 comma 1 comma 2 comma 5 comma 6. So that is my second formula. So I have two formulae here. First formula says that if you take an empty list and append it to something you will get that. The second formula says that if you can append x with y to give you z then if you had something before x if you had sort of cons something to x which is in this example the element 8 so I have added 8 to the list 1 2 so I get 8 1 2 then all I had to do was to add 8 to the resulting list that I had got in the previous time it was 1 2 5 6 now I simply say 8 1 2 5 6.

Now this is obviously a true formula you can look at it and see that it is true. The interesting thing about using logic is that this is a program

(Refer Slide Time: 22:35)

formula will become true. So logically that's the question I am asking can I find a substitution which will make this formula a true formula. And the only way I can make a formula true is by either the first clause or second clause. It should either match the first clause or if it matches the head of the second clause. So this we would call as the head and this we will call as the body. If you can match the head then you can move to the body.

We will come back to this terminology in the next class. I just want to finish this example now. So what do we do. We call our good old unification algorithm and we say that can you unify this with anything in knowledge base and knowledge base contains these two and we when you say unify we basically mean to the left hand side. Either the fact or the head of the statement. So you cant unify this with 1 because 1 says that append nil with something and you cannot unify nil with list called this is. Now remember that this list this is stored as cons this cons is nil. This notation this. Cons pair notation because that allows recursion to take place more easily. Now we can match this with the second clause. So matches 2 and the substitution is that a is this. If I put a as this may be keep searching them as I do that. Then x is equal to cons is nil. So that's the second thing I managed to match. I will bother to match the second part which is that of y so we will just write that y is equal to a list. You can write in cons pair it doesn't really matter. Then we will say that lets say z matches cons is sorry. So we will rename z as z1 for some reasons just to make life simple for us and we will say that this variable r now matches cons a we have already said is this z1.

So what have I done I have matched this variable with r with this whole thing here and I have just renamed z on the way here. So you should go back and verify that yes you should match this formula with the consequent and therefore according to the backward chaining rule we get a subgoal which is append I am left with only is which I will write now a cons is nil. With a list. I will just write this and now since we have replaced it I will call this z1. So we have a new goal now. That append cons is nil with z1. And I will leave this as a small exercise for you to show that the next subgoal from this is so we have in the second rule you can see cons a x and in our goal we have cons is comma nil. So a will match is and x will match nil. And likewise instead of z1 we can have z2 like that. But I will get a new goal which is append now is will go just in the first call this vanishes from the sentence and we will be left with only the is part. In the second call is will also vanish so I will be left only with the nil part. And lets say z2. And the thing is that z1 is equal to cons is z2. So you should verify that with this substitution where z1 is cons z2 eventually boil down to a goal which says is this formula true. That appending nil with a list called a list will give you some variable called z2.

(Refer Slide Time: 33:29)

Append

Consequent ← Antecedents
Atomic ← compound of atomic formulas

1. $append(nil, ?x1, ?x1)$

2. $append(cons(?a, ?x), ?y, cons(?a, ?z)) \leftarrow append(?x, ?y, ?z)$

Head Body


Goal: $append([this\ is]\ [a\ list], ?z)$ — matches 2

Subgoal ↓ $cons(this, cons(is\ nil))$

Subgoal ↓ $append(cons(is, nil), [a\ list], ?z1)$

Subgoal ↓ $append(nil, [a\ list], ?z2)$

$?a = this$
 $?x = cons(is\ nil)$
 $?y = [a\ list]$
 $?z = ?z1$
 $?z = cons(this, ?z1)$
 $?z1 = cons(is, ?z2)$



Obviously z2 is a variable it will be lead to and then the first clause will match. Append nil with x1 comma x1. Both x1 will match both a list and z2 will also match a list. So when you say this equal to this then you can see that as a consequence so let me write this in a different colour here. This will become a list will come here so this will become cons is cons a cons list nil. Which in other words its not very clear here but basically a list which contains a sorry this should be is. Is a list. So we will get this list which is z1 and when we put this into r we will get r as this is a list. I think you should work it out as an example. By the time we have finished answering yes we have two calls with rule number 2. So this is like recursive calls one call to the base clause which is append nil something. And the end of this we have got this binding that the variable r is bound to this list called this is a list. So in effect we have ended up appending two lists. So we try to show that a certain formula is true. But in the process we said yes the formula can be true provided r is bound to a list which reads like this is a list.

(Refer Slide Time: 35:38)

