Fundamentals of Artificial Intelligence

Shyamanta M Hazarika
Department of Mechanical Engineering
Indian Institute of Technology-Guwahati

Lecture-17
Procedural Control of Reasoning

Welcome to fundamentals of artificial intelligence within knowledge representation and reasoning we have looked at first order logic resolution as an inference mechanism, resolution refutation proofs and answer extraction. Having looked at resolution refutation proofs and the success we have had with resolution derivations one should not be mislead into thinking that resolution provides a effective reasoning paradigm within first order logic.

One needs to remember that entailment in first order logic is semi-decidable, that is algorithms exist that can say yes for every entail sentence. But no algorithms exist that can say no for every non entail sentences, resolution is refutation complete. That is for an unsatisfiable set of clauses some branch would contain the empty clause. A breadth-first search is guaranteed to show that the clause set is unsatisfiable.

However for a set of satisfiable clauses the search may or may not terminate. For many application in resolution one is interested in having derivations where it would be possible to eliminate unnecessary steps. This is what is done through certain strategies and simplifications which are refinement of the resolution process. We will wind up our discussion on knowledge representation and reasoning looking at such strategies and simplifications. We start our discussion with looking at an infinite resolution branch, here is a small example.

(Refer Slide Time: 03:06)

An Infinite Resolution Branch

Example

Suppose our KB consists of a single formula; showing R as a transitive relation. Could think of R(x,y): as x is the relative of y.

Rule: $\forall xyz\ [(R(x,y) \wedge R(y,z)) \rightarrow R(x,z)]$

C1. $(\neg R(x,y) \vee \neg R(y,z) \vee R(x,z))$

Given the query about existence of someone for everyone who is not a relative; the KB does not entail the query NOR its negation.

Goal: $\exists x \forall y\ \neg R(x,y)$

Negation: $\neg [\exists x \forall y\ \neg R(x,y)]$

$\forall x\ \exists y\ R(x,y)$

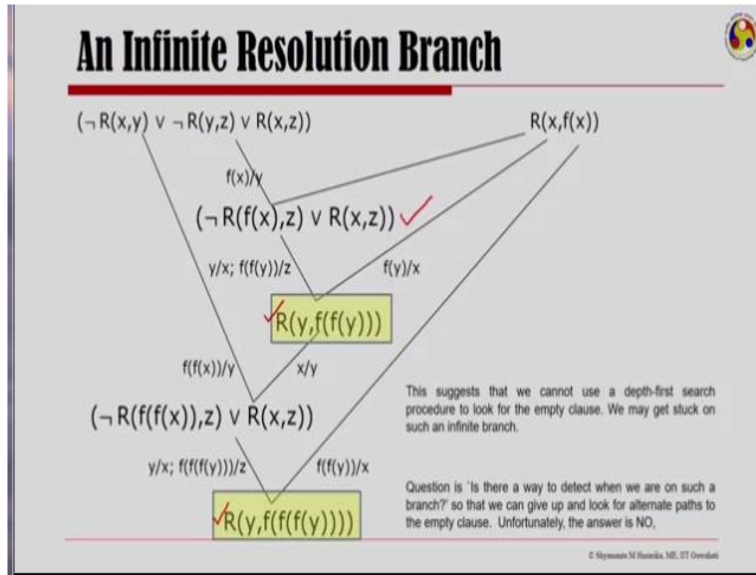
This should fail! Problem is if we pose it as a resolution, we end up generating an infinite sequence; we never get to the empty clause!

C2. $R(x,f(x))$

Let us suppose our knowledge base consist of a single formula, here this formula is showing that the relation R is transitive. One could think of R as a relation that defines relative, so R(x, y) could mean x is the relative of y. Now if x is a relative of y and y is a relative of z then we know x is a relative of z that is the rule in our knowledge base. Let us now look for the existence of someone for everyone who is not a relative.

That is let us have a query like there exist an x for all y not R of x, y, now to remind our self we will negate the goal and convert it into clausal form. So if you look at the clausal form here and the only available rule you could see that the knowledge base does not entail the query nor it is negation. So a resolution should fail, problem is if we pose it as a resolution we end up generating an infinite sequence. We never get to the empty clause, so let us look at the resolution derivation.

(Refer Slide Time: 04:44)



So here are my clauses one that comes from the rule in the knowledge base and the other that is the negation of the goal. We have a substitution f x for y and that results into the resultant that is here. Now if we resolve this further we get an expression which says R of y of f of f of y one step further resolving it we will have a disjunction and when this disjunction is resolved further remember there are only 2 clauses in the data set to start with.

So when it resolved again to the negation of the goal I have an expression which is now R of y of f of f of f of y. So if you take a minute and focus your attention on these 2 subsequent clauses that we have obtain the resolvance that we have obtain you could see that after every second step this would keep on repeating. And this suggest that we cannot use a depth-first procedure to look for the empty clause, we may get stuck on such an infinite branch.

Question is, is that a way to detect when we are on such a branch, so that we can give up and look for alternate ways to the empty clause. Unfortunately the answer is no, we are not in a position to say which of this path will take us to the empty clause. And when we should give up looking for an empty clause in a given path, for first order logic there is no way to detect if a branch will continue indefinitely.

(Refer Slide Time: 06:50)

Computational Intractability

- For FOL there is **no way to detect if a branch will continue indefinitely**
  - FOL Language is very powerful and can be used as a full programming language.
    - **Just as there is no way to detect when a program is looping; there is no way to detect if a branch will continue indefinitely.**
- Quite problematic from a KR perspective.
  - No procedure that, given a set of clauses, returns satisfiable when the clauses are satisfiable.
    - **Resolution is refutation complete**: returns an empty clause, if the set of clauses is unsatisfiable.
    - **When clauses are satisfiable, the search may or may not terminate.**

Now this is because first order logic one needs to realize is a very powerful language, in fact it can be use as a full programming language. Therefore just as there is no way to detect when a program is looping there is no way to detect if a branch will continue indefinitely. This is quite problematic from a knowledge representation perspective. There are no procedures that given a set of clauses returns satisfiable when the clauses are satisfiable.

Resolution, however is refutation complete that is it returns an empty clause if the set of clauses has unsatisfiable. And when clauses are satisfiable the search as I had mention before may or may not terminate.

(Refer Slide Time: 07:45)

Resolution - not a panacea

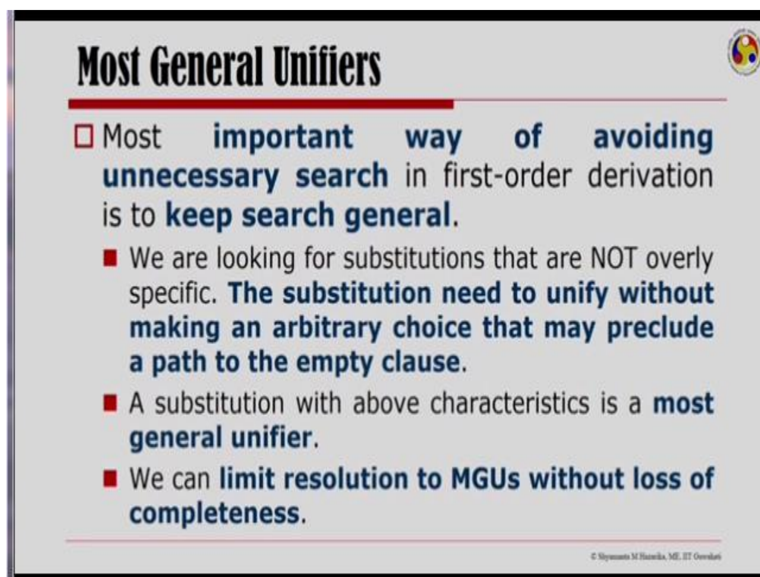
- Resolution **does not provide a general effective solution** to the reasoning problem.
  - Decision about **which two clauses to resolve** and **which resolution to perform** are made by the control strategy.
    - **Determining the satisfiability of clauses may simply be too difficult computationally!**
- Need to consider **refinements to resolution** to help improve search.
  - One option is to explore a way to **search for derivations that eliminates unnecessary steps** as much as possible.
    - We shall **focus on strategies that can be used to improve the search** in this sense.

5

Now that is why resolution is not a panacea, resolution does not provide a general effective solution to the reasoning problem in first order logic. Decision on which 2 clauses to resolve that this which resolution to perform are made by the control strategy. Now determining the satisfiability of clauses may simply be too difficult computational. So we need to consider refinements to resolution to help improve search.

One option is to explore a way to search for derivations that somehow eliminate unnecessary steps as much as possible. And that is what we will focus on today, look for strategies that can be used to improve the search in this sense.

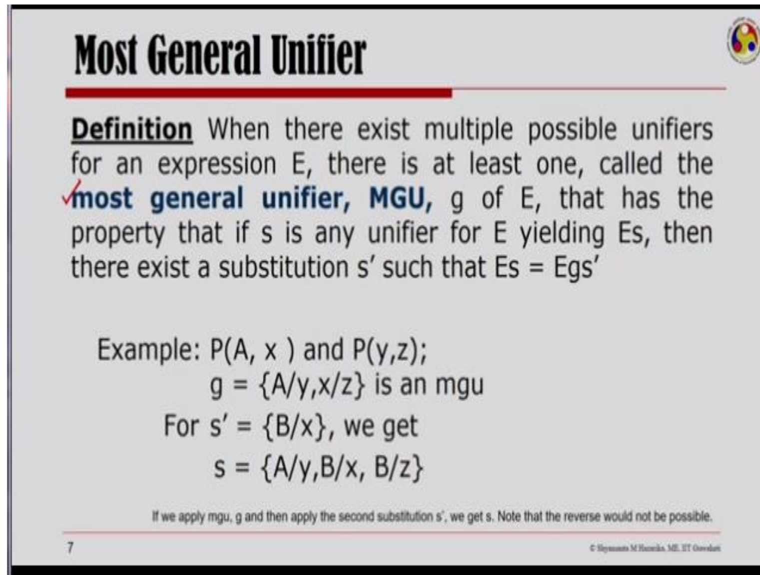
(Refer Slide Time: 08:42)



In fact the most important way of avoiding unnecessary search in first order derivation is to keep the search as general as possible. And this is achieved in the unification step by taking help of what is called the most general unifier. So we are looking for substitutions that are not overly specific the substitution need to unify without making an arbitrary choice that may preclude a path to the empty clause.

A substitution that we are looking for is through the most general unifier, we can limit resolution to most general unifiers without loss of completeness.

(Refer Slide Time: 09:35)



Now let us recall what we meant by most general unifiers, when there exist multiple possible unifiers for an expression E. There is at least one called the most general unifier which is g here that has the property that if s is any unifier for E yielding Es, then there exist a substitution s prime such that Es = Eg of s prime and then g is said to be the most general unifier. Let us take an example to understand this, so here are 2 expressions P (A, x) and P (y, z).

Now if you are looking for an unifier here then A for y and x for z is a most general unifier. This is because I can always have a substitution s prime which is B for x and then we get s which is A for y, B for x and B for z. Now if we apply g and then apply the second substitution s prime we will get the substitution s. Note that the reverse would not be possible and therefore g is called the most general unifier.

(Refer Slide Time: 11:08)

Most General Unifier

- The **MGU preserves as much generality as possible** for a pair of formulas; by using the MGU we **leave maximum flexibility for the resolvent** to resolve with other clauses.
- The **most general unifier is not necessarily unique.**

  Example P(A, x,) and P(y,z);

  {A/y, z/x} is also an mgu.

8 © Shyamanta M Hazarika, ME, IIT Guwahati

Now when I have a most general unifier, the most general unifier preserves as much generality as possible for a pair of formulas. And using a most general unifier we are actually leaving maximum flexibility for the resolvent to resolve with other clauses. So the idea is that having a general search would lead me to the empty clause, one needs to however remember that the most general unifier is not necessarily unique.

(Refer Slide Time: 11:55)

Most General Unifier

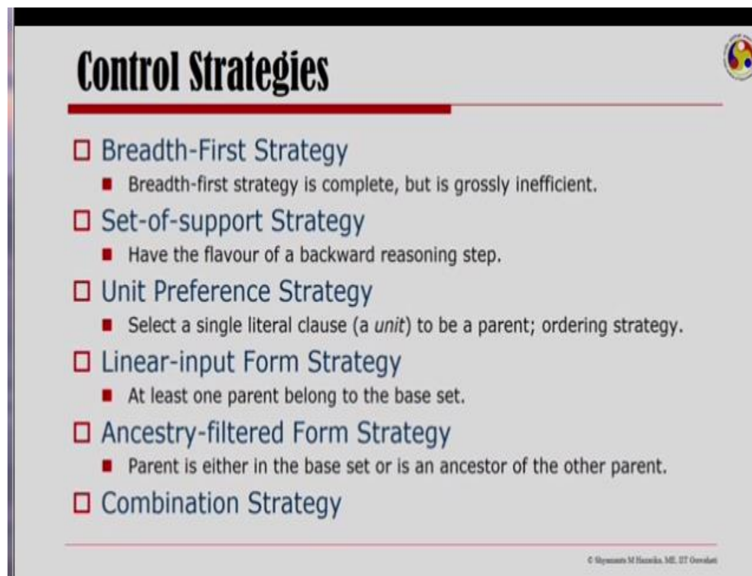
- MGUs helps immensely in search as it **dramatically reduces the number of resolvents** that can be inferred from two input clauses.
- There exists **procedures including linear time algorithms for efficient computation of MGU** for a pair of literals.
  - MGUs greatly reduce the search and can be calculated efficiently; Consequently, **all Resolution-based systems implemented to date use them.**

© Shyamanta M Hazarika, ME, IIT Guwahati

The most general unifiers helps immensely in search as it dramatically reduces the number of resolvents that can be inferred from 2 input clauses. Now there exist procedures including linear time algorithms for efficient computation of most general unifiers for a pair of literals. Most general unifiers greatly reduce the search and also can be calculated efficiently. Consequently all

resolution base systems that have been implemented today use the most general unifiers. Let us now focus on the control strategies that could be use for better search during resolution.

(Refer Slide Time: 12:44)



So the first of them that we will look at is the breadth-first strategy, now the breadth-first strategy is complete but is grossly inefficient. We would then look at a strategy called the set of support strategy, the set of support strategy have the flavor of a backward reasoning step and we will see that it explores or it generates less number of resolvents. The unit preference strategy is actually an ordering strategy and it selects a single literal clause to be a parent and takes a very interesting decision on which clauses to resolve next.

The linear input from strategy that we will discuss next is one in which at least 1 parent belongs to the base set. Then we will look at a strategy called ancestry filtered form strategy where the parent is either in the base set or is ancestor of the other parent. We will then mention about a couple of combination strategy.

(Refer Slide Time: 14:02)

An Illustrative Example

Example

1. Whoever can read is literate.
2. Dolphins are not literate.
3. Some dolphins are intelligent.

Prove: Some who are intelligent cannot read.

Predicates - R(x) : x can read.
L(x) : x is literate.
D(x) : x is a dolphin.
I(x) : x is intelligent.

Nils J. Nilsson, Principles of AI; Chapter 5, Pages 162-163.

Let us take an illustrative example to understand these strategies, now this illustrative example is a worked out example from the book by Nilsson and principles of AI, chapter 5 pages 162 to 163. These are the sentences number 1, whoever can read is literate, dolphins are not literate, some dolphins are intelligent. We need to prove some who are intelligent cannot read, let us first make our choice of predicates. So I have predicate R(x) to say x can read, a predicate L(x), x is literate, D of x, x is a dolphin, I of x, x is intelligent.

(Refer Slide Time: 15:04)

An Illustrative Example

1. Whoever can read is literate.
$\forall x[R(x) \rightarrow L(x)]$ — C1. $\neg R(x) \vee L(x)$
2. Dolphins are not literate.
$\forall x[D(x) \rightarrow \neg L(x)]$ — C2. $\neg D(y) \vee \neg L(y)$
3. Some dolphins are intelligent.
$\exists x[D(x) \wedge I(x)]$ — C3. $D(A)$, C4. $I(A)$

Prove Some who are intelligent cannot read.
$\exists x[I(x) \wedge \neg R(x)]$
Negation $\neg \exists x[I(x) \wedge \neg R(x)]$
$\forall x[\neg I(x) \vee R(x)]$ — C5. $\neg I(z) \vee R(z)$

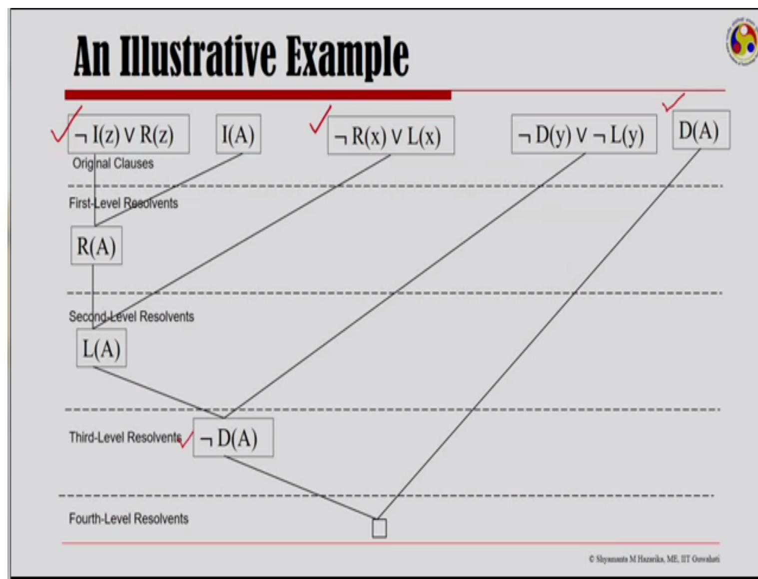
Now whoever can read is literate, I could write this first order sentence for all x, Rx implies Lx and here is the clausal form of that sentence not of Rx or Lx. The next statement is dolphins are not literate, so this could be written as for all x dolphin x implies not literate x. And the clausal

form is not Dx or not Lx, now in order to keep the variables different in all the clauses I would write not of the y or not of Ly.

Some dolphins are intelligent is written as there is an x dolphin x an intelligent x, now the clausal normal form of this would be that I would introduce a Skolem constant A and there is a conjunction n, so I would have 2 clauses D(A) and I(A). Some who are intelligent cannot read that is the goal, so we have the goal here saying there exist an x intelligent and not read. Now in order to prove this using resolution refutation I would take the negation of the goal which is not there exist x, I of x and not R of x.

And I would finally end up having the clausal form not I of z or R of z, so we have 5 clauses and now let us look at the refutation proof for this.

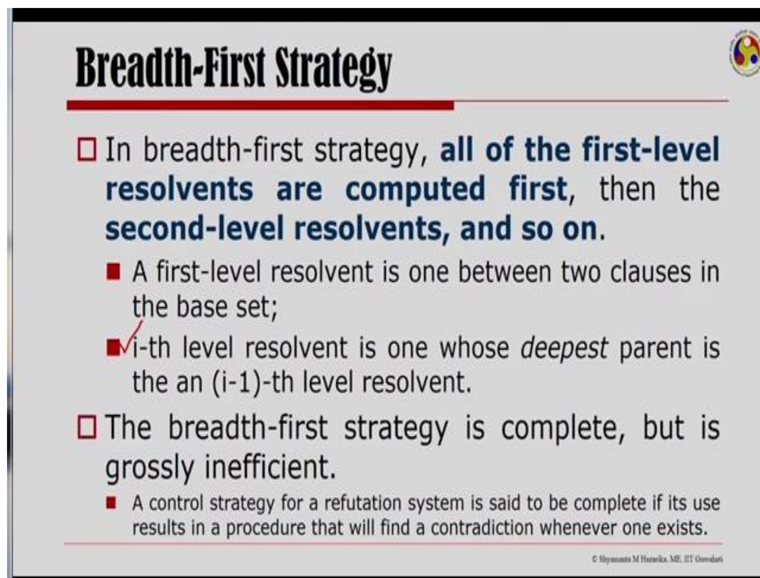
(Refer Slide Time: 17:01)



So here are my clauses, I then start with resolving from the goal well formed formula itself the clause that come because of the negation of the goal. And I have R of A that resolves with this clause here to give me L of A which then can be resolved with the clause not Dy or Ly to give me not D of A. And now once I have not D of A and D of A in my original clauses I have the empty clause and therefore the statement is proved.

Now with this illustrative example let us try to look at the different strategies that we have identified for discussion here today. And try understand how does one arrive at the empty clause in each of the strategies. Now, one needs to realize when I am using this proof that the proof of these example had taken me from the original clause to the first level resolvents to the second level resolvents, I generated the third level resolvents and I got my empty clause only at the fourth level.

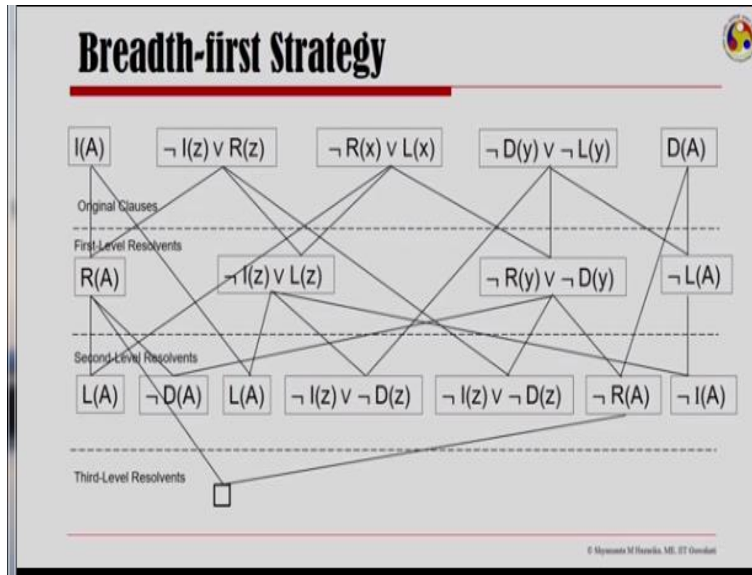
(Refer Slide Time: 18:40)



Let is now look at the breath-first strategy, in breath-first strategy all of the first level resolvents are computed first then the second level resolvents are computed thereafter the third level and so and so forth. A first level resolvent is one between 2 clauses in the base set, in fact I define something i-th level resolvent and i-th level resolvent is one whose deepest parents is an i – 1-th level resolvent.

So the breath-first strategy is complete but is grossly inefficient, now one needs to understand that when I say a control strategy for a refutation system is complete. It is in the sense that it is use results in a procedure that will find a contradiction whenever one exists.

(Refer Slide Time: 19:43)



So let us take our original problem with these 5 clauses and look for the empty clause but while doing so we would love to generate all of the resolvents at every level. So we first have the original clauses, we resolve them to generate our first level resolvents, so here are our 4 first level resolvents I(A) resolve it not of I(z) or R(z). So I have an R(A) then here I have a complementary pair R(z) and not R(x), so I could resolve them, resolve this and this to generate my second resolvent at the first level, I generate my third resolvent at the first level and my fourth resolvent.

So I have my first level resolvents once I have generated the first level resolvents I then try to resolve these with other clauses in the knowledge base. And get my second level resolvents as marked here, so one thing that needs to be noticed here is that for each of the resolution that I am performing at least one parent is from the first level resolvents set. So these are my second level resolvents and then once I have my second level resolvents I start generating my third level resolvents.

So one interesting thing that now we can notice is that I get an empty clause in a breath-first search in the third level resolvents at itself. Whereas if you remember in the proof of the illustrative example the refutation tree that I had, had to at least move to the fourth level resolvents to get to the empty clause.

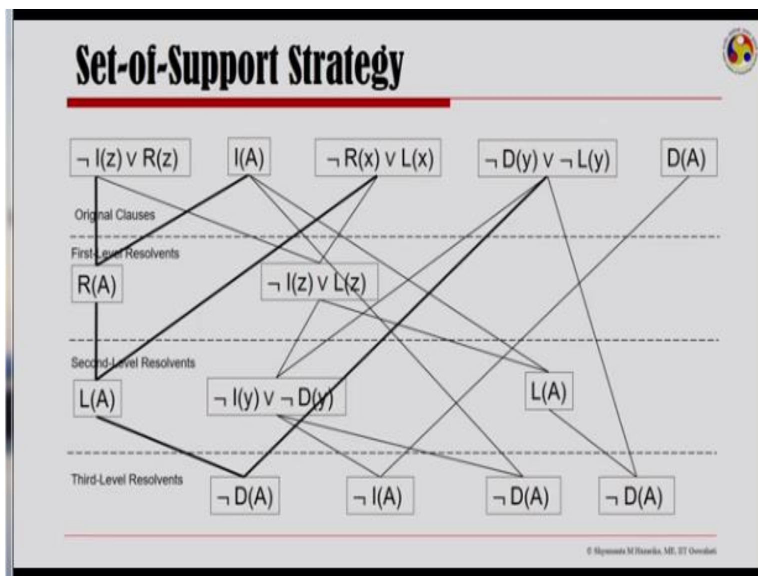
(Refer Slide Time: 22:08)

Set-of-support Strategy

- Set-of-support refutation is one in which **at least one parent of each resolvent** is selected from **among the clauses resulting from the negation of the goal wff or from their descendants.**
- In a set-of-support refutation, **each resolution has the flavour of a backward reasoning step.**
  - It uses a clause originating from the goal wff, or one of its descendants.
  - Each of the resolvents might correspond to a subgoal!

Next let us focus our attention on another of the strategies called the set of support strategy. Now a set of support refutation is one in which at least one parent of each resolvent is selected among the clauses which resolve from the negation of the goal well-formed formula or from their descendants. So in fact it has the flavor of a backward reasoning step because you are resolving with clauses that results from the goal well-formed formula or from descendants of the goal well-formed formula. So each of the resolvents might actually correspond to a subgoal.

(Refer Slide Time: 23:00)



Let us look at set of support in our illustrative example and the 5 clauses now in set of support I should resolve using clauses that result from negation of the goal well-form formula or it is descendants. So I have to start resolving from this which is the negation of my goal, so in the

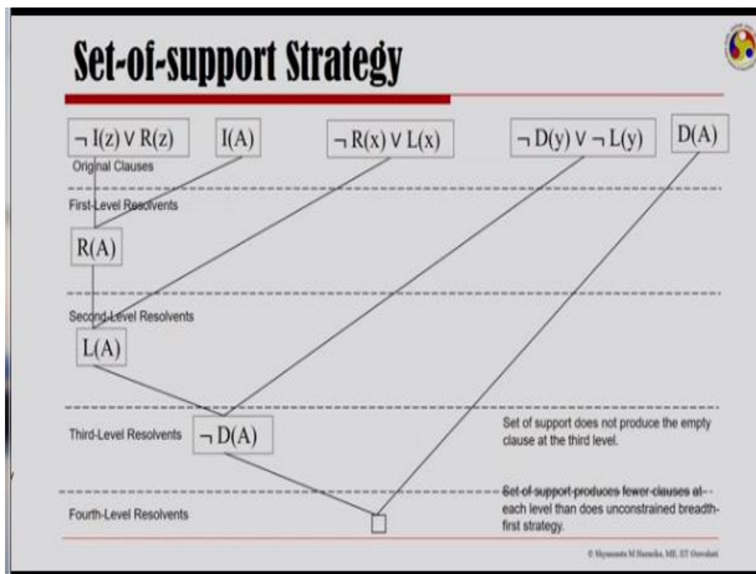
original clause set I start resolving from there and resolve with I of A to get R of A. Now the next is that this statement which is descendant of the goal well-form formulas resolution step is to be use or the goal well-form formula itself could be use.

So I take the well goal well-form formula and resolve with its second possibility here and then I resolve R(A) with not of R(x) or L(x). So I have an L(A) now I resolve this descendant that has come from the goal well-form formula with a clause from the original clause set. And have not of I(y) or not of D(y) and then I could resolve this with the original clause set to have not to have L(A), I have my second level resolvents.

And then I can think of generating my third level resolvents which is not of D(A), now one thing to observe here is that in breath-first strategy that I had shown immediately before this. We had arrived that the empty clause while we were generating our third level resolvents. But the set of support strategy was unable to get to the empty clause in it is third level resolution. So third level resolvents do not generate in this case the empty clause in contrast to the breath-first strategy.

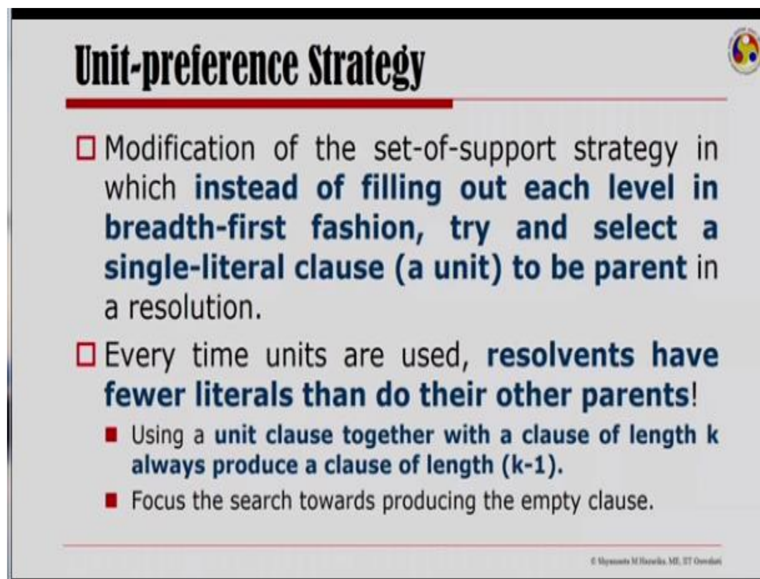
So let us see which was the path taken by my initial proof of the illustrative example. So the darken edges are the one which have been a part of the initial refutation proof.

(Refer Slide Time: 25:47)



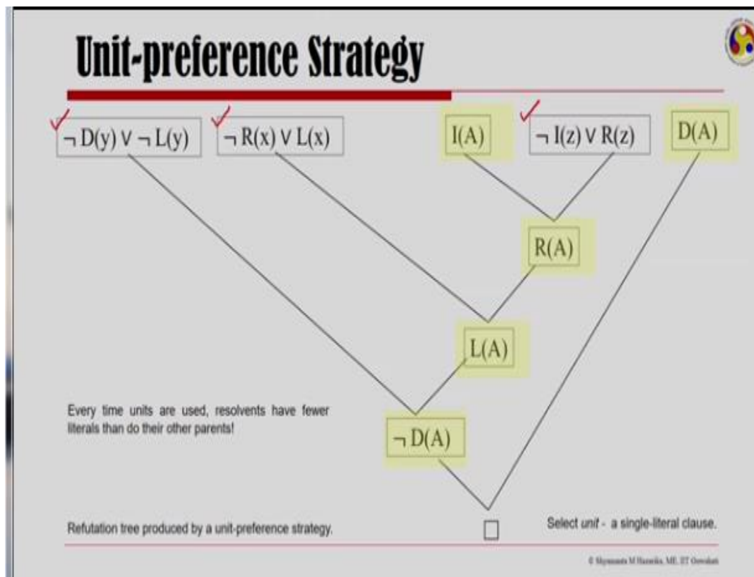
Now if you focus here attention only on the set of support and the proof that I have obtained you can see that the set of support produced the empty clause at the fourth level was unable to get today empty clause at the third level for this problem. However, it did produce fewer clauses at each level then the unconstraint breath-first strategy.

(Refer Slide Time: 26:18)



Now let us look at an ordering strategy like well I was looking for the modification of the set of support instead of feeling out each level in breath-first fashion what if one could try and select a single literal clause, that is a unit to be a parent in a resolution. Every time such units are use resolvents would have fewer literals then do their other parents. So using a unit clause together with a clause of length k would always produce a clause of length k – 1 and then you could focus the search towards producing the empty clause.

(Refer Slide Time: 27:04)



So the original refutation tree that has been produce for this illustrative example has the flavor of unit preference strategy. Let us select a unit literal clause first, so it could be my I(A) that is a single literal clause, a single unit you resolve with the complementary at this point. And generate R of A, R of A is again a unit so you select R of A and resolve with its complement in this clause, so you have L of A, you resolve with this clause which has a complement of L of A.

And you arrive at a D(A), now D(A) also is a single literal clause and in the base set you have a D(A) and you could arrive at the empty clause. Now the unit preference strategy every time the units are use you could see that the resolvents have fewer literals then do their other parents. And by just having an order on which clause to select and making a very interesting rule that you will select only the single literal clauses could take me to the empty clause without generating number of clauses as was required in breath-first or set of support.

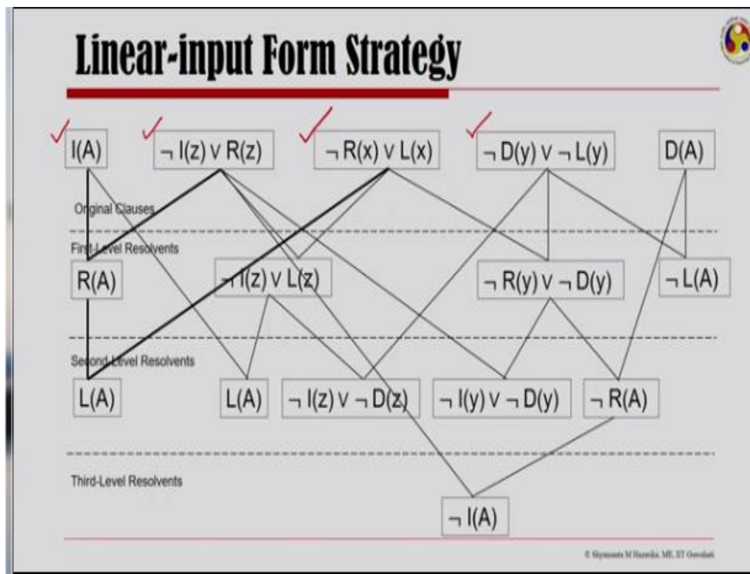
(Refer Slide Time: 28:44)

Linear-input Form Strategy

- A linear-input form strategy is one in which **each resolvent has at least one parent belonging to the base set.**
  - First level resolvents are same as a breadth-first search.
  - At subsequent levels, a **linear-input form strategy does reduce the number of clauses produced.**
  - Linear-input form strategies are not complete.

The next strategy that is our focus here is the linear-input form strategy, the linear-input form strategy is one in which each resolvent has at least one parent belonging to the base set. The first level resolvents therefore are same as a breath-first search, at subsequent levels a linear-input form strategy does reduce the number of clauses produced. Now one interesting thing to note about linear-input form strategies that they are not complete like in the sense the breath-first strategy is complete.

(Refer Slide Time: 29:24)



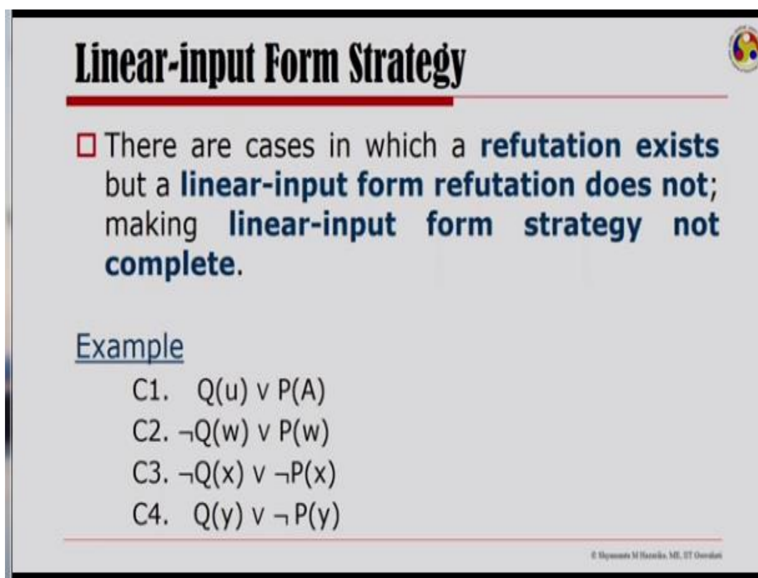
So here are my clauses from the illustrative example and let us try a linear input form strategy here, recall that the first level resolvents because everything is in the base clause I would generate as good as the breath-first strategy. But thereafter I would generate fewer clauses

because I would have to have parents which are in the base set. So I take R and here is a clause in the base set that has a complement of R, so I have L of A.

Thereafter I have I of z or L of z and I look for a complement which is I of A in the base set, so I have my second level resolvent. Here there is a complement of L in the base set so I have not of I(z) or not of D(z) and here I have a complement of R here is not R and here is R, so I would have here not of I(y) or not of D(y). So these are my second level resolvents and then I look for third level resolvents by looking at again the clauses in the base set.

Now if you recall the original proof for this illustrative example here the darken edges is a part of the original proof.

(Refer Slide Time: 31:23)



Now, one needs to understand that there are cases in which are refutation exist but a linear input from refutation does not exist. So this makes linear input from strategy not complete, in order to understand that let us take an example here. So here I have 4 clauses Q of u or P of A, not Q of w or P of w, not Q of x or not P of x, Q of y or not P of y.

(Refer Slide Time: 32:04)

Linear-input Form Strategy

The **set of clauses is clearly unsatisfiable**; but no linear-input form resolution exist.

For a linear-input form refutation, one of the parents of the empty clause must be a member of the base set.

To produce the empty clause in this case, **one must either resolve two single literal clauses** or **two clauses that collapse to a single-literal**.

None of the base case members meet these criteria.

¬Q(x) v ¬P(x)
Q(y) v ¬ P(y)
¬Q(w) v P(w)
Q(u) v P(A)
¬P(x)
¬Q(w)
P(A)

In spite of their lack of completeness, linear-input form strategy is used because of their simplicity and efficiency.

Now if you look for a refutation of these set of clauses you can see that you have set of clauses clearly unsatisfiable. The refutation tree on your right shows that I could arrive at the empty clause and therefore the set of clauses is clearly unsatisfiable. But then one needs to realize that I cannot have a linear input form resolution for these example. Now for a linear input form refutation of this example one of the parents of these empty clause that I have here need to be a member of the base set.

For that to happen one must either resolve 2 single literal clauses or 2 clauses that collapse to a single literal. Neither of these conditions are met by the base case members and therefore this example even if it has a refutation does not have a linear input form refutation. And is enough to show that linear input form strategy is not complete. However, in spite of their lack of completeness linear input form strategy is used because of their simplicity and efficiency.

Let us now focus our attention on the last of the strategies that we are discussing which is called the ancestry filtered form strategy.

(Refer Slide Time: 33:54)

Ancestry-filtered Form Strategy

- In this form of refutation, **each resolvent has a parent that is either in the base set** or that is **an ancestor of the other parent.**
- Much like the linear-form strategy.
- Control strategy guaranteed to produce all ancestry-filtered form proofs is complete.
  - **Completeness is preserved if the ancestors that are used are limited to *merges*.**
    - *Merge* is a resolvent that inherits a literal each from the parent such that this literal is collapsed to a singleton by the MGU.

In this form of refutation each resolvent has a parent that is either in the base set or that is an ancestor of the other parent. So this is much like the linear form strategy. Control strategy is guaranteed to produce all ancestry filtered form proofs is complete. So completeness is preserved in ancestry filtered form strategy if the ancestor that are used are limited to merges. Now recall that merge is a resolvent that inherits a literal each from the parent such that literal is collapsed to a singleton by the most general unifier.

(Refer Slide Time: 34:43)

Ancestry-filtered Form Strategy

The refutation tree on the right could have been produced by an ancestry-filtered form startegy.

Here the clause ¬ P(x) is used as an ancestor.

¬Q(x) v ¬P(x) | Q(y) v ¬ P(y) | ¬Q(w) v P(w) | Q(u) v P(A)

¬P(x)

¬Q(w)

P(A)

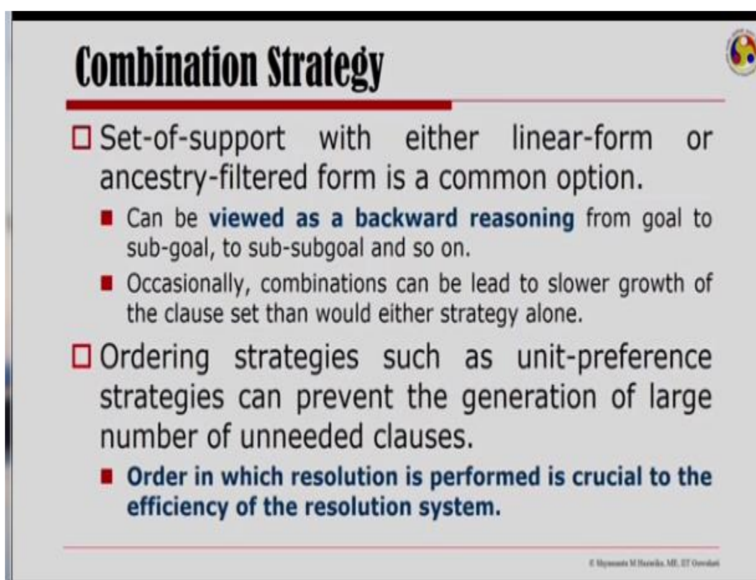
□

So let us take the example that we have use for the linear input form strategy and look at the same refutation proof towards the empty clause. Now you could see that the refutation tree on the right is produce by an ancestry filtered form strategy. Because here this clause not of p of x

could be considered an ancestor while I am resolving for P(A), I am looking for the parent of Q which is P of x.

And therefore the clause P of x is used as an ancestor and this is the ancestry filtered form strategy. Now there could be strategies that are combinations set of support with either linear form or ancestry filter form is a common option. You can view that as backward reasoning from goal to subgoal to sub-subgoal and so on. Now occasionally combinations can be let to slower growth of the clause set.

(Refer Slide Time: 36:06)



Then would either strategy alone, one interesting point to note at this time is that the ordering strategies of which we have only discuss one the unit preference can prevent the generation of large number of unneeded clauses. So order in which resolution is perform is crucial to the efficiency of the resolution system and therefore the unit preference strategy has it is own importance.

(Refer Slide Time: 36:41)

Simplification Strategies

- **Clause Elimination**

  Idea is to keep the number of clauses generated as small as possible, without giving up completeness. Exploit the fact that **if there is a derivation to the empty clause, there is one that does not use certain types of clauses.**
  - Pure Clause
  - Tautologies
  - Subsumed Clauses
- **Procedural Attachment**

  Evaluate – interpret a literal by attached procedures.

Now let us look at a couple of simplification strategies, we will discuss 2 simplification strategies one the clause elimination and the other procedural attachment. The clause elimination strategy the idea is to keep the number of clauses generated as small as possible without giving up completeness. So we exploit the fact if there is a derivation to the empty clause there is one that does not use at least 3 type of clauses one the pure clauses, 2 the tautologies and 3 the subsumed clauses.

So we could eliminate this and still be able to reach the empty clause, procedural attachment on the other hand is about evaluating or interpreting a literal by attach procedures and coming down on the size of the clauses, let us look clause elimination first.

(Refer Slide Time: 37:48)

Clause Elimination

- **Elimination of Tautologies**
  - Any clause containing a literal and its negation (i.e., a tautology) may be eliminated.
    - Any unsatisfiable set containing a tautology is still unsatisfiable after removing it, and conversely.
- **Elimination by Subsumption**
  - A clause $\{L_i\}$ subsumes a clause $\{M_i\}$, if there exists a substitution `s' such that $\{L_i\}s$ is a subset of $\{M_i\}$.
  - Examples:
    - P(x) subsumes P(y) v Q(z)
    - P(x) subsumes P(A)
    - P(x) subsumes P(A) v Q(z)
    - P(x) v Q(A) subsumes P(f(A)) v Q(A) v R(y)

So we have 2 very important eliminations one is the elimination of tautologies and the other elimination by subsumption. Any clause which contains a literal and its negation that is a tautology maybe eliminated. Now this is because any unsatisfiable set which contains a tautology is still unsatisfiable after removing it and conversely. So you could eliminate tautologies to reduce down search. Elimination by subsumption is more interesting.

We look for a clause Li which subsumes a clause Mi if there exist a substitution s prime such that L of s is a subset of M. Let us take a couple examples to understand this, here is P of x it subsumes Py or Qz, P of x subsumes P of A, P of x subsumes P of A or Q of z, P of x or Q of A subsumes P of f of A or Q of A or R of y.

(Refer Slide Time: 39:07)

Clause Elimination

- **Elimination of Tautologies**
  - Any clause containing a literal and its negation (i.e., a tautology) may be eliminated.
    - Any unsatisfiable set containing a tautology is still unsatisfiable after removing it, and conversely.
- **Elimination by Subsumption**
  - A clause $\{L_i\}$ subsumes a clause $\{M_i\}$, if there exists a substitution `s' such that $\{L_i\}s$ is a subset of $\{M_i\}$.
  - A clause in an unsatisfiable set that is subsumed by another clause in the set can be eliminated without affecting the unsatisfiability of the rest of the state.
    - Leads to **substantial reduction in the number of resolutions to find refutation.**

So if you have sums clauses that subsumes another clause in the set then it can be eliminated without affecting the unsatisfiability of the rest of the state. So these eliminations lead to substantial reduction in the number of resolutions to find refutation.

(Refer Slide Time: 39:25)

Procedural Attachment

- It is possible and more **convenient to evaluate the truth value of literals**; than to include these literals, or their negations in the base set.
- **`Evaluation' refers to interpretation of the expressions with reference to a model.**

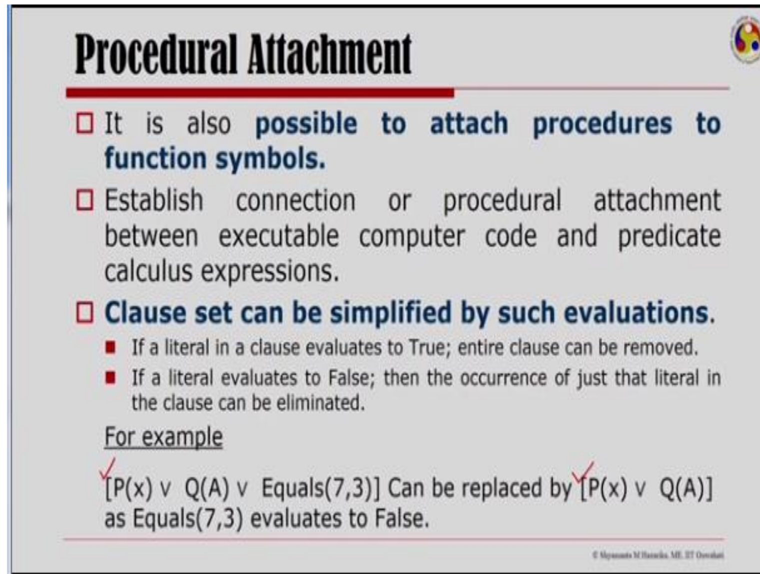
For example

Equals(7,3) can be evaluated by *attaching a procedure* that computes / checks the equality of two numbers. Given such a program for the above predicate, Equals(7,3) evaluates to False.

Procedural attachment is another possibility, in fact the more convenient one where the idea is to evaluate the truth value of literals rather than to include these literals or their negations in the base set. Now here evaluation refers to interpretation of the expression with reference to a model, so for example I could have a expression saying equals 7, 3. Now this can be evaluated by attaching a procedure that somehow computes or checks the equality of 2 numbers.

So given such a program for the above predicate equals 7, 3 evaluates to false.

(Refer Slide Time: 40:18)



So it is possible to attach procedures to functions symbols as well and then one needs to establish connection or procedural attachment between the executable computer code and the predicate calculus expressions. How does the clause set gets simplified by such evaluations, there is a 2 step process if a literal in a clause evaluates to true then the entire clause can be removed. If a literal evaluates to false then the occurrence of just that literal in the clause can be eliminated.

For example P of x or Q of A or equals 7, 3 can be replaced by simply P of x or Q of A as equals 7, 3 evaluates to false.

(Refer Slide Time: 41:15)

Sorted Logic

- Sorted Logic involves **associating sorts with all terms.**

  Example

  Variable x might be a sort **Female.**

  Function mother may be of sort **Person → Female.**
- Keeping a **taxonomy of sorts** can help.

  Example

  Woman is a subsort of **Person**
- Refuse unification between P(t) and P(s) if s and t are from different sorts!
  - Only meaningful (with respect to sorts) unifications can lead to the empty clause.

This is what is procedural attachment other ways and means of reducing search includes a technique of using sorted logic. Now sorted logic involves associating sorts with all terms, for example a variable x might be a sort female. A function like mother maybe of the sort person map to female then one would keep a taxonomy of sorts. Like one could say woman is a subsort of person, the idea is to refuse unification between literals Pt and Ps if s and t are from different sorts.

Now this under their assumption that only meaningful with respect to sorts, meaningful unifications can lead to the empty clause. And this is how sorted logic can be used to reduce down search.

(Refer Slide Time: 42:19)

Connection Graph

- Given a set of clauses, precompute a graph with edges between any two unifiable literals of opposite polarity and labelled with the MGU.
- Resolution procedure than involves selecting a link, computing a resolvent clause and inheriting links for the new clause from its input clauses.
  - No unification is done at run-time!
- Here, resolution can be seen as a state-space search problem – find a sequence of links that ultimately produce the empty clause.
  - Techniques for improving state-space search can be applied.

There is another trick of using what is called a connection graph, if I am giving a set of clauses I can precompute a graph with edges between any 2 unifiable literals of opposite polarity and labeled with the most general unifier. Now resolution procedures that can be use involve selecting a link computing a resolvent clause and inheriting links for the new clause from its input clauses, one needs to realize that there is no unification at run-time.

This if you give us bit of thought can realize that can be seen as a state-space search problem, where the idea is to find a sequence of links that ultimately produces the empty clause. Now any techniques that I can use for improving state-space search can be applied here as well.

(Refer Slide Time: 43:22)

Knowledge Representation and Reasoning

- We have discussed Knowledge Representation and Reasoning in this Module of the course.
  - Argued why LOGIC is the first choice for knowledge representation and reasoning.
- Examined FOL as a knowledge representation formalism.
  - FOL is not the only choice.
  - FOL is simple and convenient one to begin with!
- Looked at Resolution and Resolution Refutation Proofs.
  - Resolution Derivations – symbol level operation leading to knowledge level logical interpretations.
  - Answer extraction.
  - Strategies and Simplifications leading to refinements in Resolution to help improve search.

We have so far in these couple of lectures discussed knowledge representation and reasoning, argued why logic is the first choice for knowledge representation and reasoning. We have examined first order logic as a knowledge representation formalism. Now, one needs to understand that first order logic is not the only choice. However first order logic is simple and convenient to begin with, we have looked at resolution and resolution refutation proofs.

We have particularly looked at resolution derivations, symbol level operations leading to knowledge level logical interpretations. We have gone through answer extractions and today we have covered strategies and simplifications leading to refinement in resolution to help improve search. This is where we windup knowledge representation and reasoning, thank you.