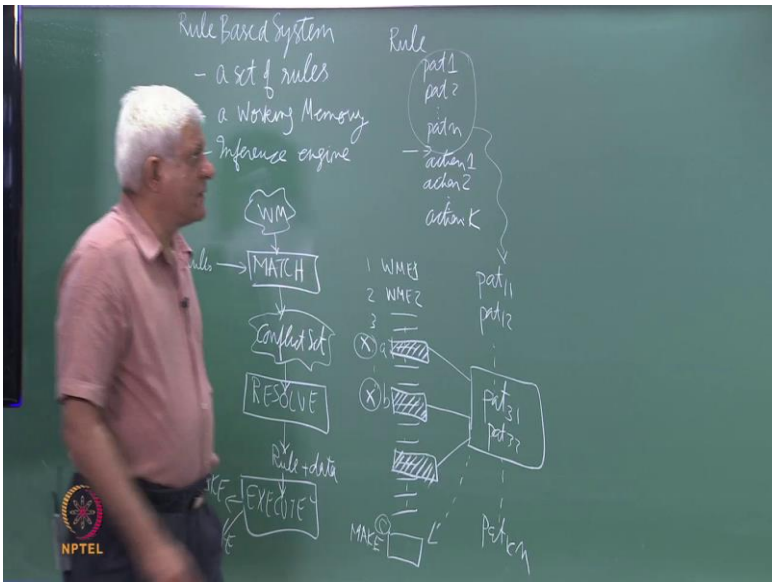
Artificial Intelligence
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 32
Rete Algorithm

.Let us begin; we are looking at rule based systems. As we discussed earlier, in rule based system, the program part, if you want to call it, is written as a set of rules. Now, these rules can do all kind of things. They can do what you would do in normal programming, for example, sort an array, and things like that, but they are also, by instrumental, in capturing what we would say as expert knowledge, essentially. So, we catch a domain expert and try to get knowledge out of expert, and put it in the form of rules. So, you could mix up rules of different kinds into one system. That is why, in the 70s and 80s, this was known as the domain of expert systems. It was quite in exercise to get to experts and elicit their knowledge from them. There was a whole exercise of what they use to call as knowledge acquisition where, they have protocols of how to talk to expert, and how to extract knowledge from them, and then put it in the form of rules, essentially. Eventually, as I said in my last class, this technology, all this has stabilized into something, which is more popular among business users, because users in banks and other such organizations; they are able to express what they call as business rules in a very simple language, and the rest of the programs takes care of everything else, essentially.

(Refer Slide Time: 01:59)



If you remember that a rule based system, consists of set of rules what we call as a working memory, which basically, holds the data and an inference engine. Inference engine is a program, whose job is to select rules, and apply them to data, and repeatedly do that, essentially. So, we saw that what this does is that it is a process, three stage process. We have match; what match does? It takes inputs. So, this is the most important part of a rule based system; what it does is it takes a working memory as an input, and it takes the set of rules as input, and produces what we call as a conflict set. So, what is this doing? If you remember that each rule is of the form pattern 1, pattern 2, pattern n, and some actions on the other side. The actions were, as we discussed earlier; either, making new data elements or deleting some. So, if you look at these patterns, this is what is used for matching the data. If you collect together, all the patterns from all the rules, then you would have pattern 11, pattern 12, pattern, let us say, 31, pattern 32. So, we will have a whole list of patterns, and this working memory is an ordered set; ordered, in the sense of time stamp. So, all these are working memory elements.

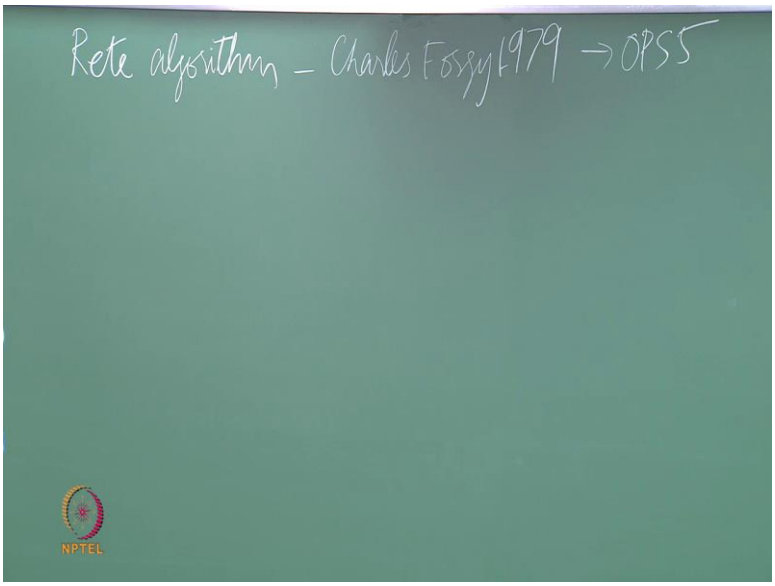
In the match phase, because we want to be complete; every pattern must be compared with every working memory element, because you want to produce all instances of rules which are matching; not just one instance. So, every pattern has to be compared with every working memory element, and that is to be done for this. So, it is a human less amount of task to be done. What this produces is a conflict set. The conflict set is a list of rules or instances of rules, along with a time stamp, like this 1, 2, 3 and so on; time

stamps of the data. Then, the task is to select which of those matching rules, we have to fire. We discussed a few conflict resolution strategies, like specificity, and recency, and mean sense analysis, and so on. So, this is given to an section called resolve, which selects one rule from the conflict set, along with its data and then, this is given to rule plus data, and by data, we mean the working memory elements; execute.

If you remember, the effect of execute is either, to make, which means to create new data elements, or to delete, or remove. These are the two principle effects we are interested in. Others, like read, print and halt, are immaterial at this point of view, and then, this is repeated into a cycle, again and again. So, we have observed that this brute force match has two deficiencies; one there is possible that different rules may be looking at the some of the patterns, they are using, may be shared, essentially. So, for the upper, this pattern 1 may be there in many rules; pattern 2 may be there in many rules, and so on, but here, we have listed them as, you know; patterns of rule 1; patterns of rule 2; patterns of rule 3; and so on. So, they are matched multiple number of times. They may be even shared partially, as we will see in an example today. That even, if they are shared partially, we should be able to minimize the number of matches that we do. Remember that these patterns are made up of class names and attribute value pairs. For every attribute, there is a test condition, which could be equality that it must match the data, or it must be something, like greater than a value, or not equal to a value and things like that.

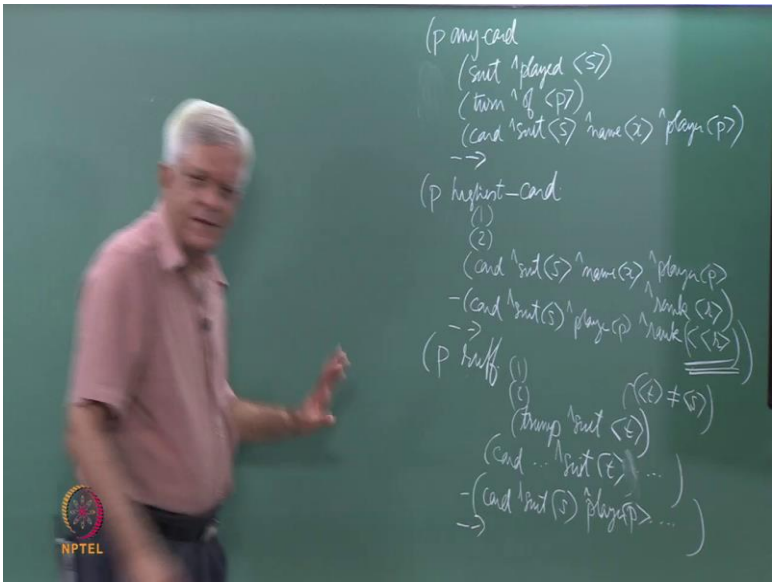
So, that is one thing that we would like to do. The other thing that you would like to do is that, because when we, let us say this, some rule 3 is selected in the resolve phase, and let us say this rule 3 matches these pieces of data; let us say 3 pieces of data. It is possible that when this rule 3 executes, it may say, delete this data. So, let me put a cross here; delete this data and may be, it might say, add a new data, or let me use a term; make here. So, let us just take this simple example. We are deleting two elements. Let us call these elements, A and B, and you are adding a third element, which let us call as C, essentially. That is the only change you are making into a working memory. Two elements have been removed and one has been added. So, what we expect in such a situation is that most of the other rules, which are matching with some data, earlier in the last cycle, will continue to match in the next cycle. We would like to avoid the trouble of matching them all over, again.

(Refer Slide Time: 09:23)



So, these two properties of this particular match algorithm are done away with by the algorithm that we are looking at today, and it is called the Rete algorithm. It was, as I mentioned earlier, given by Charles Forgy in 1979 as a part of his PHD work. Based on this, he divides the language called OPS5 which, some people say, stands for Official Production System Language. Remember that, we also called rules as productions, and all this was happening in CMU at that time. Today, we want to look at the details of this algorithm; the Rete algorithm and its structure. So, let me also rewrite the rules that I wrote in the last class. So, remember this card playing rules, we have said.

(Refer Slide Time: 10:18)



This is an OPS5 syntax; something like this. If you remember, the first element in this list is the class name of the data structure; the second element which, with this symbol is the attribute name; the third element is the value of the attribute name, this attribute; and you can have multiple attributes and their values. When we enclose something in angular brackets like this that represents a variable, essentially, which means, it will match anything. There are no data types involved here. Let us say, we also have a working memory element, which says, who has to play. So, this one says, it is a turn of some player P, of course, this is a variable. So, it could be applied to any player, and card suit s name X, and we are saying, in this third pattern that there is a card, which is also; whenever, we have same variable name in two different patterns in the same left hand side of the rule, they must match identically, to the same value, essentially. They cannot match different values. So, if there is a card of this group is played, whose name is X and is held by player P, then this player can play this card.

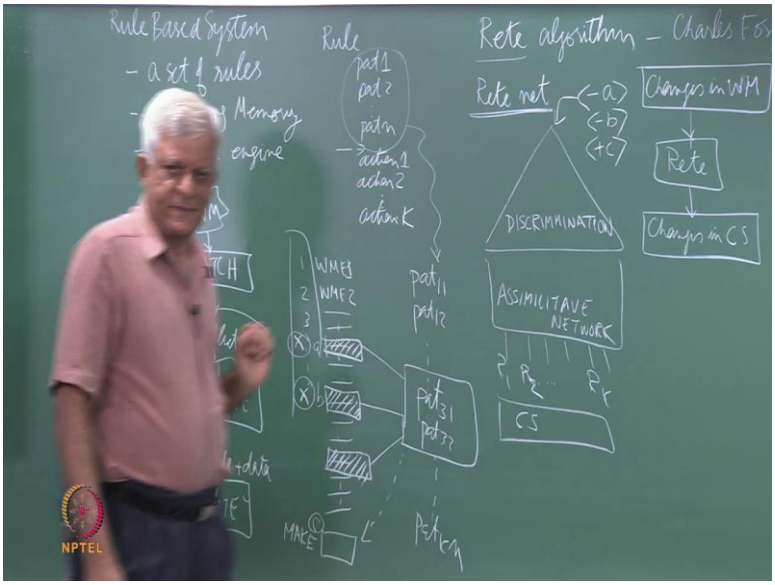
I will not write at the right hand side, because today, we are only interested in the match part of the algorithm. Then, we had written another rule, P highest card. So, instead of playing any card, now, I have a different rule, which says that you know, in that suit, play the highest card. So, this 1 and 2 will be the same; I will not repeat that here, but this is 1 and this is 2; it is just for our sake, but not part of the language syntax. So, anyway, let me rub it off from here. The first was same, the third one is similar card; it must be of this suit, s; it must have some name, because you want to use that name in the output, say, play this card, and must be held by this player, whose turn it is to play, and this card has a rank, let us say R. Now, we are interested in playing the highest card. So, we have this extra bit of information in this particular pattern. Obviously, notice that patterns can select any subset of the class information.

I put this in bracket, just to make it show; this is the condition that we are looking at; yes, and this is the negation sign. This should be right as saying, the first two conditions are the same that the suit being played is s; it is a turn of player P. Player P has a card in this suit s whose, rank is R, and there is no card held by player P, in this suit s whose, rank is smaller than R, which means higher than R; we assume that. Then, we will say, play this card. So, in both instances, we will play this card name s of this suit; play this jack of spade, something like that. So, let me just give you a third rule. Those of you are literate, will recognize as playing a trump card in a, when you do not have a card like this. So, the

first condition is the same; the second condition is the same; we have a third condition, which is saying that from trump suit t. So, those of you have played cards, would know that there is something called a trump suit, and essentially, the effect of calling a suit, a trump suit is that its cards are always, higher than other suit cards. But they can only be played, when you do not have the other suit cards. Again, we are talking about a card here. So, I will not write this name and all, but what I will write here is that the suit is trump. Everything else will be the same that there is a card, held by player P, whose name is X, and blank; we are not bothered about here, but this suit that I am selecting is this trump suit T, and not the suit s, which I have said here.

I can even, to be explicit, I can even say, instead of writing just T, I can write t not equal to s, but it turns out that in this particular example, that does not really matter; so, I have not written it like that, but those are the kind of things you can write as condition checks, essentially, and no card. So, this is the another root, which says that if the suit in place s; if the turn of the player is P, is there; the trump suit is T, and this player P has a card of the trump suit, and he does not have any card in the suit, which is in play; then we can play the trump suit. So, just three different left hand sides that we are interested in, and we will see how Rete net handled it. So, Rete net is the structure, which is maintained by the Rete algorithm or Rete algorithm, as some people call it. Rete is the latin word, which actually, means net, essentially.

(Refer Slide Time: 17:20)



The Rete net is made up of two layers. You might think of it as on the network. On the top, is a discrimination network, and we will see what the structure is. Below, is an assimilative network. So, the top part of the network is what some people called as a mainly sorted, decision key or discriminative network. The bottom part collects together; this is updated. So, what do you mean by assimilative; that for every rule, I need a set of patterns to match, and the bottom parts will see that it gets those three patterns matched, essentially. How does this work? This basically, works by inserting at the top, these three tokens. So, when you say you want to delete this working memory element, we generate a token; let us say we call it minus a; minus stands for the factor we want to deleting it, and one for this token b, which is minus b, and one for this token c, which is plus c.

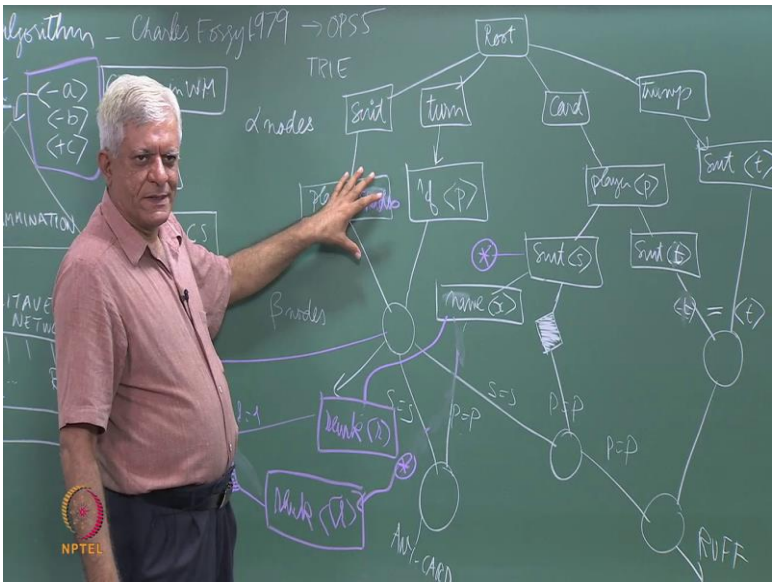
So, as the algorithm is progressing, every time it fires the rule, it will generate some positive tokens and some negative tokens. Those tokens would be inserted here, at the top of the network, which is, as I said in the last class, changes in the working memory. These tokens capture what has changed in the working memory; which things have been removed from the working memory; what things have been added to the working memory. What the Rete algorithm does is that takes changes into working memory and produces changes in a complete set. So, here, are the different rules R1, R2, Rk; so, the network, which you will see in a little bit more detailed in a moment, is a compilation of the rules, essentially. This network is defined by the rules that we have in the system, and it is basically, a different representation of the same rules in a network form. The working memory, this part, sits in locations, which are nodes in this network, as we will see in detail. So, the working memories, kind of, distributed over this network, and you can imagine that tokens are flowing from top to down, just like you would do, for example, in the very simple case of binary search key; you want to find out, whether there is a record whose, key value is 17. You will put it at the top, and you will check, whether that root node is greater than 17 or less than 17, and it will send it down one branch. You will keep doing that. Eventually, it will filter down to the node, whose value is 17. In our case, also we are looking for tokens with, of a specific kind, but what we are looking for, tokens with match our rules, essentially.

So, our tokens will follow down this path, and for example, in the first two, we need three tokens. So, three different tokens, if they come there, this part is only to make the search efficient, just like in a search tree. This part is to collect together, the different

pattern set or rule need. So, if you can somehow, collect three tokens for this tool one, and the three tokens arrive here, then we will say rule one is matching. Once it is matching, and as long as these tokens are sitting there, it will continue to match, essentially. Only, when we make these changes, if something happens, then it may go out of the conflict set or something like that.

So, these rules, this gives us the conflict set, here somewhere. Remember, conflict set is a collection of instances of these rules. Each rule may have more than one instance with different pieces of data. Collection of instance of this rule, along with the identifiers of the data elements; they are matching, which is the time stamps that we use here. So, these are the identifiers. They basically, tell you when this token meet was. So, to start with, you can imagine that whatever the data that you have, you just put the whole thing into the net. Instead of just having a few changes, the whole working that is a starting point, and it will go and figure out different set of rules; you will get the conflict set. Then, the resolve face will select one of those rules in the conflict set; execute that; generate a few tokens like this. Those few tokens would be put in here, and they will again, tickle down this network, and make some changes on the way, but otherwise, the rest of the match is, sort of, static in these networks. So, let us see what this network looks like. As I said, the top part is discriminative in nature, which tries to separate tokens of different kinds. So, that you do not have to do a sequential match. That is a basic idea in many of these search algorithms. The first thing, we test for, is a class name. So, we have one class name here, suit.

(Refer Slide Time: 23:16)



So, let us say this is the root and there is a node; first nodes are called alpha nodes. Just some nomenclature; nothing to do with the game playing algorithm that we have been looking at. And what alpha nodes do is that they have exactly, one parent and they do a certain test, and the test is what we are writing in the box, and in the first level, it is a class name; that the class name is suit here. They may have more than one child, depending on what the rules are saying. For example, here, we have for the card, some different rules. So, you may have more children, essentially. So, the second one is for some attribute from the class name. Now, in our case, the attribute is played and our value is only a variable. So, we do not really have much, but you can imagine that if we had one rule for spades, and another rule for hearts, then you would have one branch for spades here, and another branch for hearts here, and this test would be more specific. You could say if the suit being played is spades or the suit being played is hearts, then the rest of the network would be different.

In our example we have only a variable. So, it does not matter. We have only one. Then, let us finish up the easier path first, which is the second last name, which is turn, and it has also, only one value; see, in all the rules, we have only one value; turn of P; turn of P. Again, you can imagine if we discriminated between the different players, if we said turn of south, or turn of north, or turn of east; where, north, east, south, west are the names of the place, then we would have different branches coming out of here. So, again, we have only one branch, because there is only one pattern of this kind. Every path in

this represents a pattern. If we have different patterns, which are slightly different, then they will diverge, you know; one pattern will go this way, and another pattern will go that way. So, it is like a structure, which is similar to the Trie structure; I do not know if you have studied the Trie structure.

It is a little bit like that, essentially. So, all these are alpha nodes. Alpha nodes have one parent, and every node has a memory, associated with it in which, tokens can sit, essentially. So, let us assume that memories, behind it, and we cannot see properly. Then, other kinds of nodes are beta nodes. I will draw the beta notes with the circle, and what they do is; pull together, tokens from different patterns; tokens, which are of different patterns, essentially. In our example, there is no test to be done. These are kind of independent and all we are saying is that we should have one token of this kind, and one token of this kind. Then, we have these two patterns, at least. Our first tool needs three patterns; we have collected two. So, these are beta nodes. Beta nodes may have or in the implementation that we have described here, have exactly two parents. We can have more than two, but let us assume that this is like a joint of two structures. We are only joining two patterns at a time, essentially. So, if we have three patterns, then we will first join two, and then, the third one, and so on. Beta node also, may do a test as you will see in the moment. So, let me do this rough rule first, at this end. Obviously, one thing we need is card, because that is one of our class names, and the last one that we need is trump. It has one attribute called suit, and it has the value called t. So, this last rule needs five patterns; one and two, which we have here. Then, one pattern will come down this path, which is that trump path, and then, two patterns will go down this path. So, two patterns are talking about cards, essentially.

So, one of them says that player; I have not written that there, but it is a part of that pattern, and this P must match this P, as you will see. So, card player P suit t. So, all we are saying now is that if this player has a card of this suit t, then he is allowed to play. So, we joined together, these two patterns by a beta node, and we put a condition that this t, which is coming from this side, and this t, which is coming from this side, equal to. So, we are using the equality condition, because we are just identifying that; see the variable names I have used the same, but they do not have be the same. There could have been different variable names, but this test is saying that in this case, it is the same. So, let us see.

Let us say I use this variable s here, and then, I will have to say this s equal to t, or let me just keep to simple thing; see, this is t. We will look at the other one in a moment. So, it has got four, but four these things, but it still needs one pattern, which says that it does not have a card of this suit s. Now, we want to do a match, which says that he does not have, or such a data element, does not exist. Now, that can be done in a various number of ways. Remember that we said that every node is associated with a memory, essentially. Now, I will just use some symbol, which I will claim is a symbol for negation. The way to read this is that the token will flow down this path, only if the count; now, this is a bit weird; only the count of the number of tokens, passing down this path, with this value P and with this value S, is 0, essentially. So, you must have; I am just trying to use the simple way of talking about it, essentially. What you want to really say is that if a token of this kind want to actually, come here, then its effect on the rule will be that it will not match, essentially. You can think of it like that. You can think of it as saying that the count of the number of tokens. What are these tokens? This is the pattern, which says that is the card held by player P of suit S. Now, if the player had three cards of this suit; let us say we are talking about spades.

If the player had three cards of spades, then three tokens would come down here, essentially, because that should be in the working memory. What this test is doing is counting, how many tokens are coming here, and if the tokens is greater than 0, then it will say, no, this rule cannot fire. The other way to look at is that if a token wants to come here, then it will immediately, disable the rule to which, it is pointing to. So, in any case, we need a joint here, and the joint should say that this S is the same as this S; so, S equal to S. I will just write, and this P is the same as this P. So, by this moment, we have looked at three patterns. There are two positive patterns and one negative pattern. In that branch, we have looked at two patterns; both of them are positive patterns. If all the five patterns match, which means, the appropriate tokens come down here, then we can join them together, and we have to be careful here. So, P is equal to P, for example, and that is the only thing, you have to worry about, and here is a rule called ruff.

So far, we have drawn the network for only one rule, which is the third rule. The third rule has five patterns; this one and two are same as this; that the suit will play is that the turn of player P. The third one says that the trump suit is t. The fourth one says that player has a card of t, and the fifth one says that the player has no cards of the suit, which

is being played S, essentially. So, this snow card is captured here, because of this symbol.

This captures the first two patterns, and this is the third pattern, which is the trump suit, which defines a trump suit. This is the fourth pattern, which says that the player has a card of this trump suit, because T equal to T, and here, all the five patterns will come and sit here, and we will say, this rule is now, gone to the conflict set. Let us take the first tool, which is the simple rule. All it says is there is a card of suit S, name X, player P. So, I could simply say it here. I could say here, I need one more thing which is named. So, if a pattern wants to come down this path, it would say that it is the card held by player P variable P, some suit, which is a variable S, and some name, which is variable X, and I can join it with this, and put all the conditions; that S equal to S, which means S coming from this side, is the same as the S coming from this side, likewise, for the P coming from this side, P equal to P and I would have that any card.

Now, you can observe that the only thing I did at this moment, for the rule number one was to add this box here, and this beta node here. This is the alpha node and this is the beta node, and nothing else, I have to do. The rest of the match, it is the first pattern; the suit being played is S; the second pattern, the turn of player P is already there, in the network. I have to just take a lead from there, and I have already joined them together. Then, take this third token from here, and at this point, I have three tokens coming together, which is what I need from a first rule, essentially. So, this intra cycle savings, so to speak, that in the same cycle, I am matching this rule and this rule, but I am sharing in this network; the matching work, which is required for those rules. Only, some things are different, which I do separately, essentially. You can take the middle rule, as you can see, it always takes something from here, and it needs a card of suit S name X player P rank Y. So, we can take this from here. Let me use a different color. So, we can take this from here, and add another alpha node of rank R. I can take still, one more from here, and add another alpha node, which you will say, rank less than R. Remember, R is still a variable at this point of time. Then, I can combine them by saying this R equal to R. So, what is happening in, if you can see this purple part? We are talking about two tokens flowing down this path up to here. Actually, we do not need this name, sorry. So, this should actually, come from here.

The card held by player of suit S, whose name is X, and whose name is R, and somebody should have pointed this out. I need this negation node here. So, in this rule, the third pattern is flowing from here; card held by player P suit S name X rank R, and its coming here. The fourth pattern is coming from here; card held by player P suit X, then it is coming here, whose rank is less than R, but then, there is a negation here, which means the player does not have a card of higher rank, and this I can combine with this.

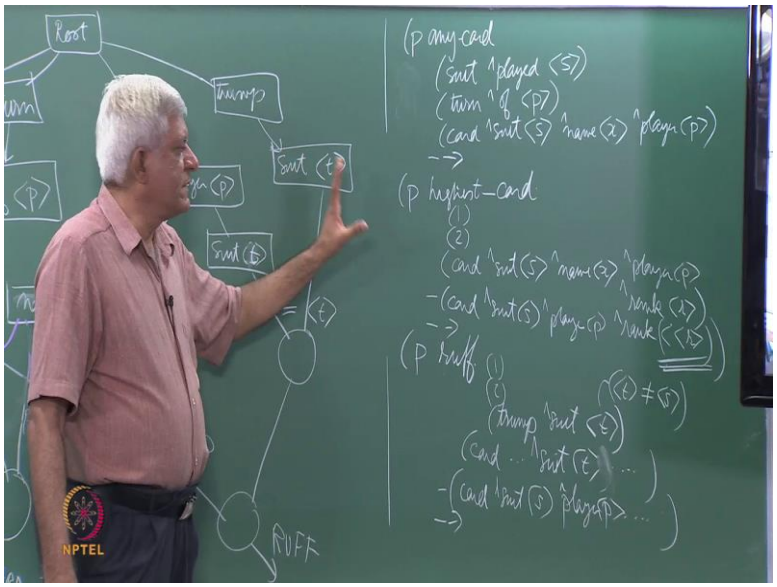
With all that P equal to P, and S equal to S, and I will get the second rule, which is highest card. In this network, see, we started off by creating network for the third rule. Then, we added few more edges and nodes for the first rule, and then, a few more edges, missing for the third rule. So, notice that there is only one branch going down for the cards being held by players, which means that every card that every player holds, a token will go down this branch. The same token, if you look at this structure here, all you are saying is that it is of suit S, and this is of suit T where, S and T are variables. So, obviously, every card, let us say it is spades. So, if there is a spade card held by this player P, the token will come here, as well as, it will also go here, but only if it matches this, it will match this rule number four; otherwise, it will not match this rule number four, and only if this matches this suit, which is in play, will the player played this card, essentially

The Rete network is a network, which is a compilation of the rules. So, if you have this network, then you do not need the rules, essentially, because it is just a different way of writing these rules. The task of the software engineer here, is to be able to view the set of rules, and construct a latter network corresponding to this, essentially. That is the first part, building the network or compiling the program, as you might say, and then, you put in all the data from the top. This act like a discrimination network, if the token working on the element is of class name, card; it will follow this path. If it is of class name, trump; it will follow that path, and that each stage, it is a test; that it has to satisfy; call it to move on. This one says that the effect is opposite, that if such a token exist, and this rule will not fire. If there is no such token coming here, then this rule will fire, essentially.

So, obviously, you can see that this needs four tokens; four positive tokens, and one negative pattern, which stands for the fact that they must be no such token. So, if a token comes down this path, that rule will get disabled. Now, we can also understand what is

going to happen in this cycle here. Whenever, we execute a rule, we add these new tokens, and put them down the network; sort of, we push them from here, and they will travel some down.

(Refer Slide Time: 42:10)

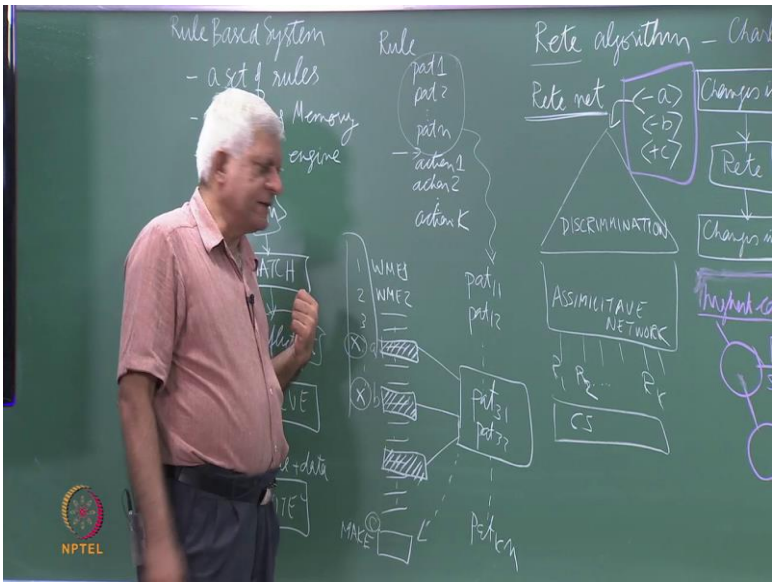


So, it is possible that we have created, only these three; let us say these two tokens, and this tokens come in later, or let us say we have these four tokens, and in the last; and let us say that the suit being played is spades, and in the last round, if the program played a spade, then we will have a negative token of saying that that card has been played by the suit, and if that player had only one card of the suit, there is the negative token, will come here. It will cancel the positive token, which was sitting earlier, and then, this rule will certainly, come into the conflict set. Essentially, the effect of throwing these tokens down, this network is to either, activate some new rules or to deactivate some existing rules, essentially.

Any questions at this point? So, I will expect that if I give you a set of rules of some domain like I define, you can look at some past papers, for example, you should be able to construct the network, and show, where the tokens are. Of course, here, we have not created the tokens. So, we do not know where the tokens will sit, and here, every test is a very variable kind of a test. So, it will not stop tokens here, but supposing, this was played spades; let us say this rule was only for spades. So, if I had spades here, then if a token with hearts or diamonds or clubs come, it will just sit here. It will never be able to

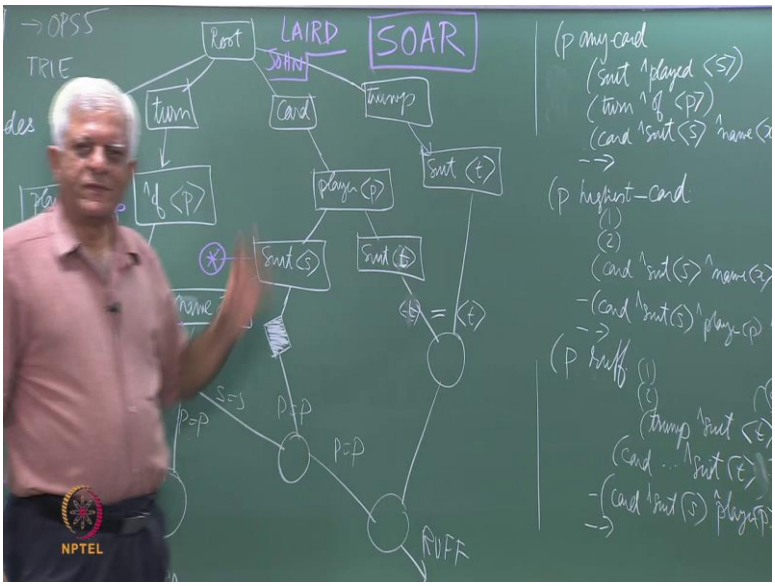
reach this place, because there is no rule to tackle those kinds of processes, essentially. So, we should able to say where the tokens are sitting, essentially. So, as the last bit, we started off by saying that this whole idea of rule based systems was motivated by a cognitive approach to problem solving.

(Refer Slide Time: 44:40)



We are also said that the rules, for example, are the long term memory of the problem solver, because that is the knowledge, which is acquired over a period of time, whereas, this working memory elements are the short term memory of the problem solver, because that is the data that the problem solver is interacting with. There are cognitive models of the brain, which says that this is where, you store short term data, and this is where, you store long term data. So, in some sense, this is the model for doing this kind of reasoning. There are variations of this, we can think of. For example, you can think of a language in which, you can create rules on the fly, essentially. That would be difficult to handle in this, because here, the network that we have described is a static network, and we have said it is a compilation of the rule that we have; how do we handle a situation where, the rules can be themselves, added or deleted from a knowledge base?

(Refer Slide Time: 45:42)



Finally, I should add that you should look up system called Soar, which is a successor of this language, which amongst, apart from this whole idea of using a Rete network; the Rete network has become like a fixture in all these business, I said, last time we have mentioned that. You know this Rule Business Management System or Business Rule Management System; BRMS. So, there is lots of software, which is commercially available for managing business. Rete networks or its improvements; I also said that 4G created a newer version of Rete network, which is called Rete NT, which is 500 times faster than this network that we have just described, but unfortunately, we do not have descriptions of that network, because it is a trade secret, and it has not been revealed, but they have been several improvements, in terms of speed, that have been made to the Rete network. It is a big part of business rule, in general, essentially. This Soar, on the other hand, was continued in the university environment, again CMU, by a guy called John Laird.

So, if you look up John Laird or if you look up Soar, you would get more information out of it. John Laird also made an observation about four or five years ago. He said that we all know that the computing industry is virtually driven by games, essentially. I mean, most of the faster processors, for example, are developed, because people want faster games; you should be able to do things faster. But what John Laird observed was that games, or at least, the graphic capabilities of games, you know the ability to render scenes, which are realistic, and ability to have characters moving around, which look like

realistic characters, has kind of saturated; that there is not much improvement you can do on that front, essentially. What he says is that the next level of improvement in games will be in giving a quote unquote; intelligence to the characters. So, whatever, avatar or whatever, characters we have in game; if they are intelligent, which means if they have some knowledge of some kind, and they can do some useful, what you may call, thinking; then, that is going to be the next level of interesting list in game, essentially. He is the same guy, who is maintaining this Soar architecture.

So, those of you are interested in games; I am sure, most of you are; I would encourage you to look at Soar, and if you want to build a game, which has intelligent characters inside it, then this is a mechanism; this is a language, which allows you to; which is a kind of decedent of OPS5, which allows you to express what kind of knowledge that you want put into to an agent, inside a game.

So, we will stop here with this. In the next class, I want to move to a different topic, which is that of planning. We have talked of planning on and off, but we will look at a closer look in a little while.

So, we stop here.