Constraint Satisfaction Problems

Prof. Deepak Khemani

Department of Computer Science

Indian Institute of Technology – Madras

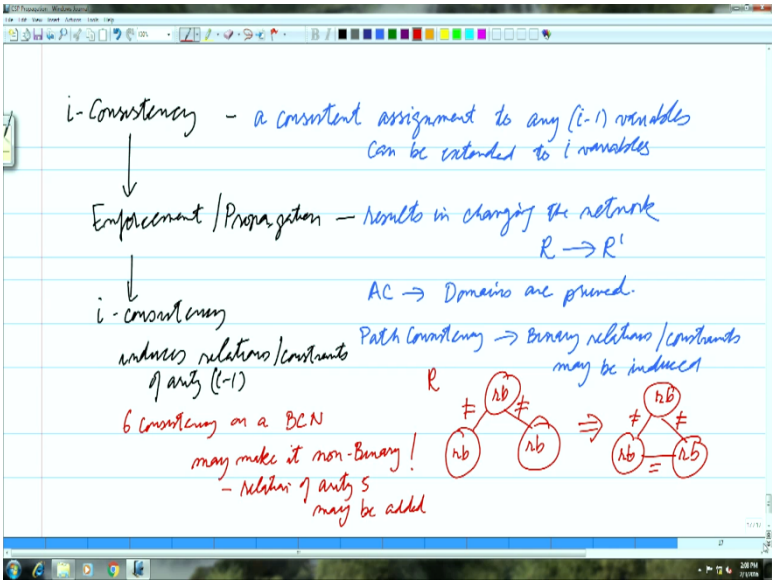
Module – 3, Lecture – 06

We've been looking at consistency enforcement and the general idea is to talk about i-consistency. i-consistency says that a consistent assignment to any i-1 variables can be extended to i variables. Arc consistency is two consistency. Path consistency is what we are looking at this moment and that's three consistency.

Enforcement of consistency results in changing the network. We use the term network and CSP interchangeably. When you start with R, you end up with some R' which is slightly different. We saw that when we do arc consistency, domains are pruned. So the domains may change. If you do path consistency, then binary relations or constraints may be induced and we saw an example of this. Let me just repeat that. If you have three variables and the relation is not equal to in the two colour map colouring problem, then we transform it to a new network. So, the third relation, the equality relation is induced.

In general enforcing i-consistency induces relations of arity i-1. We will see that this is the generalization we can make and its indeed true. So what does this mean? It means that supposing we do six consistency on a binary constraint network we may make it and I say may make it in the case when it was not originally six consistent. If it was not six consistent it may make it non-binary because a relation of arity five may be added. So this is the generalization that you must keep in mind when you're looking at specific algorithms – arc consistency, path consistency and so on. This is generally true of i-consistency.

(Refer Slide Time: 5.24)



Now, the questions that we want to often ask about consistency is, what is the impact of i-consistency? Why are we looking at this consistency at all? The two questions that we are interested in asking are can it say anything about the existence of a solution? And the other question that we often want to ask and we will do so more and more is, can we have back track free search? We will try to answer these questions as we go along but I can remind you of some examples that we have seen. By back track free search, we mean that the search algorithm never has to back track, even though the algorithm that we will study will, and it is in fact called back tracking.

What does back tracking imply? You give a value to the first variable, to the second variable, third variable, fourth variable. We looked at the n queen's problem if you remember. Then at some point you get stuck. You cannot give a value, let's say, to the ith variable. Then a typical back tracking algorithm will go back to the i-1st variable and try a new value. If this doesn't happen then the search is said to be back track free and we will see that under certain conditions, we can make the search back track free.

So, there is always going to be a trade-off between propagation and search. So, the more enforcement of consistency you do, the less amount of work the search algorithm will have to do. And ideally if it is back track free that basically it means its linear. Give the value for the first one, second one, third one and so on essentially. That's a trade-off and it's a question we'll be interested in. We saw the example of map colouring. The relation is not equal to and we have two colours. It is arc consistent but no solution exists. So, arc consistency is not enough to tell us whether a solution exists or not.

On the other hand, if you look at an example, so it is some constraint problem and whatever the relation is, it is shown by the matching diagram. What can you say about this particular network? This network is path consistent. If you go back to the definition of path consistency, it says that every edge in the matching can be extended to a triangle for every other variable. And you must keep in mind that there is an implicit constraint whenever there is no constraint stated. It exists because you're allowed to choose any of the two values. There is no constraint in that part essentially. So always when we're looking at such diagrams you must keep in mind that if there is no edge in the constraint graph, then it means the universal relation but in this example, if I look at the matching diagram, I should really draw the edge because it's allowed. This is an example which is path consistent but not arc consistent.

Why is it not arc consistent? That is because if I choose the value X = a, the solution cannot be extended. Remember this is the matching diagram. It means if there was a related value in the other variable then there would be an edge from a to that value. Which means that this is a path consistent network because every edge can be extended to a triangle. As you can see there is only one triangle here but it's not arc consistent. So, path consistency does not imply arc consistency but path consistency is enough for BCN to say that a solution exists.

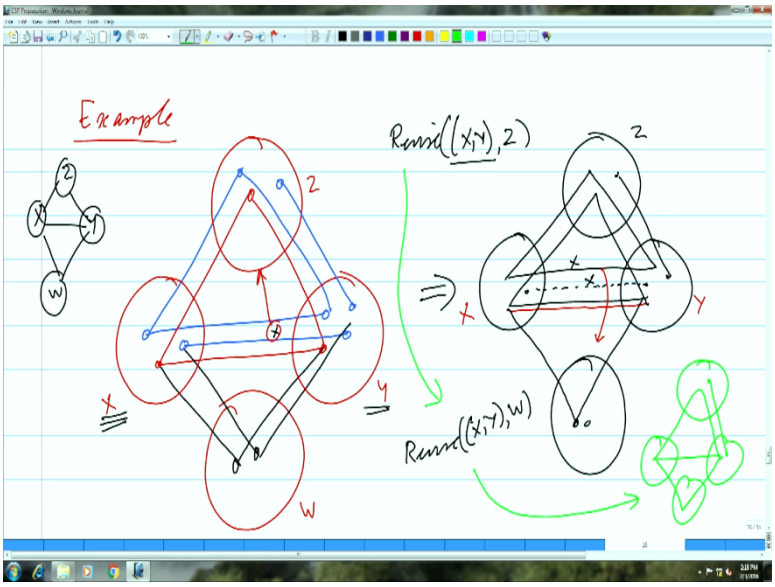
I must qualify this statement by saying that strong path consistency is enough. So, the word enough is actually tied up to the word strong because if I give you a trivial example in which nothing is related to anything else, then this is by definition path consistent. If you remember the definition of path consistency, every edge should be extended to a triangle but there is no edge here so this is vacuously true and it is by definition path consistent. So, its path consistent but it's not arc consistent and it doesn't have a solution.

We need strong path consistency to guarantee that a solution exists for a binary constraint network and in general, strong i-consistency implies that for all $j < i$, network is j-consistent. I've introduced here the notion of strong consistency and strong consistency says that a network is strongly i-consistent if it is i-consistent as well as for all $js < i$, it is j-consistent. So of these networks, you can see that none of them is strongly i-consistent. We'll keep addressing these issues as we go along.

In the last class when we were talking about the revise algorithm, we said that if we have these four variables X, Y, Z, W and we have the edges in the matching diagram as shown and we have two operations –first XY with respect to Z and then XY with respect to W. A pair of triangles would have remains after these two calls. So, after you do these you get this network

in which the edge between Z and Y would've remained. If you had one more call of revise ZY with something else, then the edge that is hanging there between Z and Y would also have been removed essentially.

(Refer Slide Time: 14.31)



So, let's look at what revise is and we can generalize it to higher order revise. I may not do it explicitly but you have seen revise one or revise as we called it. We said that for every value in the domain of X, if there is no value in the domain of Y, then delete that value from domain of X. That's what revise was doing. It was pruning the domain of X when we were revising X with respect to Y.

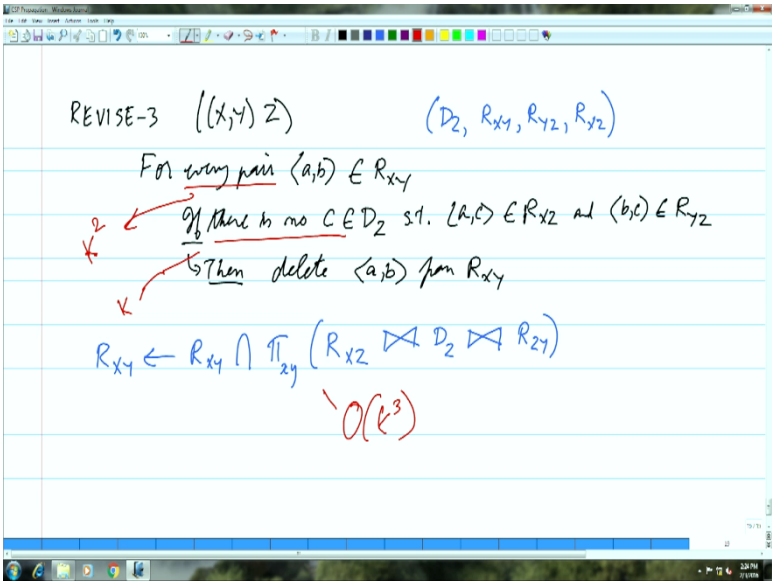
When we talk about revise three which is needed for path consistency, we can use two notations. The notation that Dechter uses is Revise(XY, Z) and the other notation which I sometimes use uses the arguments Dz and the three relations Rxy, Ryz and Rxz. The revise algorithm says that for every pair (a, b) ϵ Rxy, if there is no c ϵ Dz such that (a, c) ϵ Rxz and (b, c) ϵ Ryz, then delete the pair (a, b) from Rxy.

Essentially, we are saying that for every edge in the matching diagram, if there is no c which will make it a triangle then remove (a, b) from Rxy. This is the basic revise algorithm and it prunes the set of binary relations or it induces a set of binary relations. So if you have a universal relation, it may induce it as we saw in the example with three nodes and the map colouring example.

We can also write this using the notation of relational algebra. I'm pruning the relation Rxy. From Rxy I may remove some things or I may keep only some things – whichever way you want to look at that. What do I want to keep? Whatever I can project on to the variables XY from Rxz ∞ Dz ∞ Rzy. So, I create a set of three tuples by doing this join by selecting only those things in which there is a path going from the domain of X to the domain of Z. That's what I'm doing by joining with Dz and then from there to Y essentially. So, that's like joining whichever are the edges in between Z and Y. So, I can use this as an alternate notation for revise instead of the algorithm. Rxy = Rxy ∩ πxy (Rxz ∞ Dz ∞ Rzy).

What is the complexity of revise? When we say for every pair, there are k squared pairs and if I'm checking that there is no c, I need to look at, in the worst case, k elements. k is the size of every domain. So essentially revise is $O(k^3)$. Revise three is $O(k^3)$. Revise two or revise was order $O(k^2)$ essentially. And from there again you can jump to a conclusion – revise i would be of order $O(k^i)$ and revise i would basically say that for every set of i-1 variables if there is no value in the ith variable then remove that tuple.

(Refer Slide Time: 19.56)



Once we have revise we can talk about the algorithm and like AC-1 we have an algorithm PC-1 which is also brute force. The first thing to observe here is that we're not going to look at the constraint graph at all unlike what we did in AC-1. In AC-1 we were only pruning domains and we were only pruning those domains which were related and for universal

relations, we did not need to prune anything so it was not part of the algorithm but here we may end up introducing new relations.

So, again just to remind you, if you had the map colouring problem with not equal to, not equal to, we'll end up inducing a relation of equal to. So that means originally Rxy was universal which means anything in X was connected to anything in Y. This is not a very general problem. They had only two variables – red and blue. Only then you end up inducing the relation. If there were more colours then you would not end up doing that essentially. So, for this problem, Rxy was originally universal and we changed it to equality. We had to look at all pairs. The point is that not just for the pairs which are related, not just XZ and ZY, but we also have to look at also XY because it's from XY that we're deleting this pair. So, we have to call revise of XY with Z also even though XY is not part of the relation.

I'll write the algorithm at a high level. The general algorithm is that we'll revise for each pair of variables Xi and Xj, and not for every edge which is a different thing all together. You call revise three of Xi, Xj with Xk essentially and you do this till no relation changes. It's analogous to AC-1. It's simply a brute force algorithm. Call revise three with all possible pairs. Actually you should think of it as all possible triples since at any point you're looking at three variables at a time. If a relation changes then do the whole thing all over again. That's the brute force part of it.

What is the complexity of this? In the worst case, how may cycles will we do? In the worst case, only one edge per cycle will go. Worst case means we have chosen the wrong order of things. Only one edge is vanishing per cycle. So, in the worst case we can have $O(n^2)$ because there are n^2 pairs of variables and each pair of variables have n^2 edges between them. Then the worst case is $O(n^2k^2)$.

What is the complexity of the loop? You're processing n^3 combinations or triples of variables because you're choosing for each variable which gives you n and for each pair of variables gives you n^2. So, there are n^3 triples of variables and for each of them revise costs k^3 since revise is $O(k^3)$. So, if you put this whole thing together, then in the worst case you have to basically multiply these things – $O(n^5k^5)$.

This will give you a feel that as the amount of consistency that we enforce is bigger, in the sense that as the i of i-consistency becomes larger and larger, the complexity of the revise itself increases. It'll become k^i plus the number of combinations of variables that you have to look at – that also increases and therefore consistency enforcement becomes more and more

expensive. So we will need to slowly move towards looking at how can we do less work in consistency enforcement and still gain a lot from that.

We will continue with path consistency. Then we will talk about i-consistency briefly and we will move on to trying to see if complete consistency enforcement is necessary or not. Can we get away with partial consistency enforcement? And the motivation for that or the justification for that would come from the fact that at least the simplest algorithms will process the variables in a chosen order. The search will always say, look for a value for variable X1, then for X2, then for X3 and so on essentially. Maybe we can exploit that order to do less amount of enforcement. In the next class, we will move towards an improvement of PC-1 which is similar to AC-3 but which we just call PC-2.

(Refer Slide Time: 25.52)