Constraint Satisfaction Problems

Prof. Deepak Khemani

Department of Computer Science

Indian Institute of Technology – Madras

Module – 3, Lecture – 03

We are looking at constraint propagation and we're looking at i consistency. So let me recall the definition of i consistency. It says that any partial solution of i-1 variables can be extended to i variables. A partial solution is an assignment which is consistent with all the constraints it covers. We say a network is i consistent if we take any partial solution of i-1 variables, we can extend it to i variables.

The simplest form is 1 consistency which basically means you can give a value to the first variable, whichever you choose. And what do we mean by saying you can give a value to them? It means you can choose any value, which has an implication that a network may not be one consistent if there are unary constraints in the network. For example, if there is a variable X and you say that there is a constraint $X < 3$ and if the domain has {1, 2, 3, 4, 5}, then it's not 1 consistent because you cannot choose 4 or 5 because those values are not consistent with the constraint that it's value should be < 3.

So making a network node consistent, this is also called node consistency, making a network node consistent is simply looking at all the unary constraints and deleting whichever values are there in domains which don't respect that constraints. It's a very simple algorithm. Just look at each domain and see if there is a unary constraint which that value doesn't satisfy and delete those values.

Then the thing that we focussed on was 2 consistency or arc consistency. It says that any assignment of one variable can be extended to an assignment of two variables. What does it mean in terms of matching diagram? I've drawn the constraint graph in red. The matching diagram basically says which variables are related to which variables. Remember we are talking about the binary constraints. What arc consistency says is that for any value in a domain, it must be connected to one value in each of the domains it is related to.

The black one is a matching diagram. The red one is the underlying constraint graph and arc consistency says that for any value that you pick, there must be a corresponding value in the related domain.

So we have looked at two algorithms AC-1 and AC-3 and for every edge for both sides, at the end of the edges, they prune domains appropriately. For example I have X and Y, Dx = {a, b, c} and Dy = {1, 2, 3}, and my matching diagram says that a is connected to 2, b is connected to 2 and a is connected to 3. Then after AC, after making it arc consistent, we get Dx = {a, b} and Dy = {2, 3} because we delete the value c from the domain of X because it doesn't have a corresponding value in Y and we delete the value 1 from the domain of Y because it doesn't have a corresponding value essentially.

So that's what propagation does. It basically prunes the domains of every variable. Now you can see that pruning a domain is kind of equivalent to inferring unary relations. Now instead of saying that domains are pruned, you could have said I'm inferring a relation R on x which say that {a, b} are allowed and I'm inferring a relation R on y which says that only {2, 3} are allowed essentially. And then of course you can make it node consistent and then c and one will get pruned but the larger point that I want to make here is that whenever you achieve i consistency you infer a relation of arity i-1 essentially. So whenever you do arc consistency we infer a unary relation and when we see higher order consistencies we see that we infer appropriately the relations. We'll see this but this is a property we can easily observe for arc consistency.

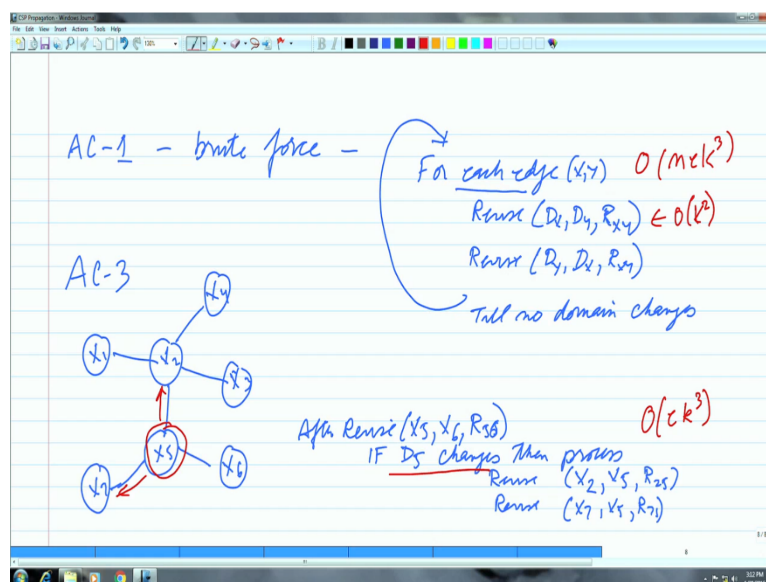
(Refer Slide Time: 8.47)

So just a quick recap of the algorithm that we looked at. We saw AC-1 which is essentially brute force. It says, for each edge X Y, revise domain of X with respect to domain of Y looking at the relation Rxy and the other way around. Revise domain of Dy with respect to Dx again looking at Rxy and do that for each edge. And you put this into a loop till no domain changes. So this is rather a very brute force process.

And then we came up with a new algorithm which was AC-3 which did not do this brute force process that in every cycle you look at each edge. What AC-3 did was that if you have, for example, some random constraint graph that I've drawn with 7 variables - X1, X2, X3, X4, X5, X6 and X7, and supposing you do revise X5 with respect to X6 with respect to the relation R56, if D5 changes, then process X2 with respect to X5 with R25.

So essentially if there is a change in X5, i.e. if D-five changes then just simply worry about its related variables and only do revise for them with respect to X5. So obviously you can see that it is much better than AC-1 because it will not call revise unnecessarily all the time and it'll only do it for those domains which have got affected. If you remember the complexities of this, it was $O(nek^3)$ cube. We had done some reasoning for this. k^2 of this came from the revise operation and the rest comes from how many times you do the looping and so on. e is the number of edges.

(Refer Slide Time: 12.59)



Now it turns out that AC-3 also may be doing too much work. Let me give an example. Supposing you have X, Y and Z and Dx = {a, b, c}, Dy = {1, 2, 3} and Dz = {A, B, C}. They

don't have to be different domains but I'm just using different names here. a is related to 1 and to 2. b is related to 2, c is related to 3 and let's say 2 is related to A and 2 is related to B and 3 is related to C essentially. So in this example, after revise Dy with respect to Dz with Ryz 1 is deleted from Dy. That means Dy has changed and this means AC-3 will call in this example revise Dx with respect to Dy with respect to Rxy.

But if you look at this example, you see that this call is not needed. Why is it not needed? It's because even after you have deleted 1 from Dy, a does not lose its support. So 1 was a support for a or 1 was related to a. So we're worried that a must have lost its support because we're deleting this element 1 from Dy. We're worried that Dx has a value a which was supported by this 1 and because we are deleting 1 it may have become inconsistent.

But if you observe, a has another support which is two. It means Dx is still arc consistent with respect to Dy and the call that AC-3 makes is unnecessary. So essentially what we want to do is to try and look at an algorithm which will maintain its data at the value level and not at the domain level. What AC three says is that if you've revised a variable Y with respect to a variable Z and the domain of Y has changed, or it has been shrunk, then any variable that was related to Y is in danger of becoming inconsistent and therefore just to be safe make a call to revise, in this example, XY. And if you remember the AC-3 algorithm it maintains a queue. It would've put the pair XY into the queue. But there are times when this is not necessary essentially as illustrated by this example and in such a situation we can try to find a more optimal algorithm.

When we talk about complexity, you can see that AC-3 is already $O(ek^3)$. A simple check whether an edge is consistent itself is of $O(k^2)$ because you have to look for each value of one variable whether there exists other value in other variable and then you've to look at all these edges at least once essentially. So can one actually find a more optimal algorithm? It turns out that yes indeed we can and that's the algorithm that we want to look at next.

That algorithm is called AC-4 and what AC-4 does essentially is keep track of the information of support for each value. If you keep track of how much support each value has, then you know whether to propagate the change. So essentially this is a propagation algorithm. We will see that whenever a value is deleted, has one of its related values in the other domain become orphaned or not. By orphaned you mean it has no support left essentially.

In this particular example, you can see that when 1 is deleted from Dy, a is not orphaned essentially. a is not orphaned and the other values are not deleted from Y. In this example only one value is deleted from Y so anything that was connected to that value was in danger of becoming support less but in this example because a has two supports - 1 and 2, a is not orphaned.

So the algorithm AC-4 that we will look at next essentially counts for each value, for each other variable, how many values are supporting it? So in this example a is supported by two values from Y, b is supported by one value from Y and c is supported by one value from Y and as and when we delete values, we will decrement appropriate counts and if a count becomes zero then we know that we have to delete that value and if we delete that value then we have to worry about whether to propagate that change or not.

So the level of granularity at which AC-4 looks at the constraint network is at the value level. It'll check if deleting a value is changing something in my network or not. Whereas AC-3 looked at constraint graph level. It said that if a domain is changed, is any connected domain likely to be affected? So it takes the worst view and says that must be affected so let's just call revise essentially.

AC-1 just looks at the network as a whole. It says, if some domain has changed, just call all revise combinations all over again. So that was really brute force. AC-3 looks only at the constraint graph level and tries to decide which revise to call and AC-4 will delve further down into the value level and then decide whether to do something about it. And we will see that AC-4 does not actually call revise because revise itself is $O(k^2)$.

Revise X with respect to Y says for every value of X, check whether there is a corresponding value of Y. So it becomes $O(k^2)$ already. AC-4 will not need to do that. It will simply say because of the value that I'm deleting, in this example 1 from the domain of Y, has a related value become unsupported? So it'll only look at the value level essentially. And as we will see this will allow us to bring down the complexity of the algorithm to $O(ek^2)$ but you must also try to imagine what is the best case complexity. So far we have been talking about worst case complexity. So whenever you say revise is $O(k^2)$, we are assuming that for every element in the first variable we will look at all the elements in the second variable. So that's why k+k+k+k… k times so that will be k^2 essentially.

In the best case for every element the first value we look at would be related so revise will become $O(k)$. That's the best case but we have been talking about worst case complexity and

the algorithm AC-4 has a worst case complexity which is better than AC-3 and we will look at that in the next class.

(Refer Slide time: 21.31)