Constraint Satisfaction Problems

Prof. Deepak Khemani

Department of Computer Science

Indian Institute of Technology – Madras

Module – 3, Lecture – 02

We have been looking at the algorithm AC-1 and this is the first of the algorithms that we'll look at for arc consistency. The algorithm basically said that for each pair xi, xj of variables that participate in a constraint do the following. Make a call for the first variable with respect to the second variable and the relation Rij and then make a call for the second variable with respect to the first variable for the same relation. We'll use the same notation for that. We'll assume that it's a symmetric kind of a relation, that (a, b) is present then (b, a) will also be present and so on. So I'm not writing Rji explicitly here.

So let's assume that we have a small constraint satisfaction problem in which there are three variables. Let's call them X, Y and Z. The matching diagram is as shown. So there is no distinction between blue and red except for the fact that you will see that the blue ones will survive and the red ones will not survive in this situation.

So the three blue edges are arc consistent in the sense that the three values always have a connected value in each of the other domains. So we're talking about three relations now, Rxy, Ryz and Rzx. So the calls that I'm interested in will be X with respect to Y, Y with respect to X. Let's say in this particular order – Y with respect to Z, Z with respect to Y and Z with respect to X and X with respect to Z. So we've these three pairs of variables. This particular example has three constraints between each of the three variables.

So when I do X with respect to Y, what will happen? All four values have a matching value in this. And then when I do Y with respect to X, then again I find that all four values have a matching value to this essentially. The next is Y with respect to Z. So there are four values and each of them has a value in Z and likewise Z with respect to Y has a values for this essentially.

Only when I come to Z with respect to X, there is a first delete and what is being deleted is the particular value because there is no matching value in X. And then when I call X with

respect to Z, I will get my second delete. I have used a star there. So that will be the one that will be deleted. So I have finished one round. Now is this one round enough? That's the question we were asking and you can see that it's not the case that it is enough because now one node has been orphaned.

So once you delete that starred value, then this particular value in Y which I've shown with a black arrow does not have a matching value in X. Now Y is no longer arc consistent with respect to X. When I've deleted the first value it has likewise made the corresponding missing orphan. Now Y is not arc consistent with respect to Z. So after the first delete, Y is not AC with respect to X and after the second delete, Y is not AC with respect to Z.

So I have done one round of every pair of variables in both directions but I end up with a network which is not arc consistent. Which means I must do some things again. So the simplest thing to do again is what AC-1 does. AC-1 puts this whole thing into a loop and it says keep doing until no domain changes.

So in our example what has changed? The domain of X has changed and the domain of Z has changed. We have deleted one value in this example from the domain of X which is marked by the star and we have deleted one value from the domain of Z which is also marked by a star. So these two values have gone. I have also circled them. Now two domains have changed and I'm not sure that my system is going to be arc consistent or not but it's not necessary that deleting a value makes the edge arc consistent.

So the two orphaned values which, let me draw a square around them here, are both, in this example, in the domain of Y. They became not AC because they have a connection only to one value in the other domain and that value got deleted. So in the next algorithm, we will try to basically propagate this change but in this version of the algorithm AC-1, we will just do a brute force approach. If something has changed just do the whole thing all over again and of course, eventually it will work.

So if you follow this example to completion, then you will see that all the red lines will get deleted. So we're assuming that a line will get deleted when a node at the edge of that line gets deleted and only the three blue nodes and the three blue lines will remain. And that network will be arc consistent essentially but it'll take a bit of an effort to arrive at that.

This algorithm AC-1 is brute force. It simply says until no domain has changed. What is the worst case complexity of this algorithm? And whenever we're talking about complexity we'll

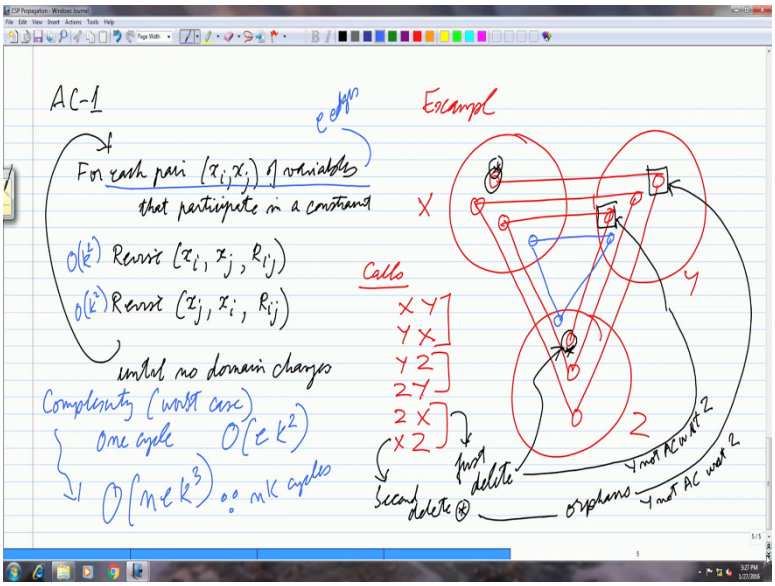
always talk about worst case complexities. Remember that there are n variables in general. Each variable has k values and there are e edges.

Now if you remember the complexity of revise is order k^2. And it is done for each pair of variables. In the constraint graph there are e edges and therefore you will do e*2*O(k^2). So the complexity of one cycle here is order ek^2. For each edge you're calling it twice but two of course as you know we ignore when you're writing complexity.

Now how many cycles will happen in the worst case? Now it turns out if I had not drawn the blue triangle, if I'd only drawn the red triangle then you can see that every variable, every value would've got deleted and in fact all the domains would have become empty if those blue cycles were not there. So this is something that I should mention. Whenever you're implementing these algorithms if at any point any domain becomes empty, you can just terminate and say that there is no solution because the fact that a domain has become empty simply means that you can't give a value to that variable and which means you can't have a solution but in the worst case all the values are going to get deleted. How many values are there? n*k values because there are n variables and k values.

So in the worst case, in every cycle, only one value will get deleted and therefore you will need n*k cycles. And therefore, complexity of this is O(nek^3) because there are n*k cycles. Why n*k cycles? Because in each cycle one value is getting deleted. And there are n*k values. So this is the complexity of AC-1 essentially.

(Refer Slide Time : 12.24)



Let's look at improvements upon this. This algorithm is called AC-3 because there was proposed an algorithm at some point called AC-2 but it turned out to be too close to AC-3 and therefore we never study AC-2. Some people say that the Waltz algorithm that we talked about in Huffman Clowe's labelling is actually a version of AC-2 or AC-3.

So what do we want to do in AC-3? Let me just go back to the previous diagram. We saw that in the first two deletes that we made here which was in the cycle of Z; making Z consistent with X and X consistent with Z, with respect to Z, we deleted two values from the domain of variable Y and its only that is what is getting threatened. That is, the values that we deleted from the domain of Y, are they making some other variable in other variable inconsistent?

So we need to only worry about that. For example, at the end of this cycle, we have not deleted any value from X or Z so whatever consistency that was existing between X and Z is not going to change. The only thing that's going to change is the consistency between Y and something else. Some other variable may have become inconsistent with respect to Y because I've deleted a value from Y and that is what we should worry about essentially.

And that is what this algorithm AC-3 does. It maintains a queue of pairs. So for every pair of variables xi xj, that participate, as before, in a constraint, which means e pairs because there are e edges in the constraint graph, you update the queue. You start with an empty queue.

So I'll just say enqueue the pair (xi, xj). So there is an initial cycle in which I add all pairs. This is exactly like we did in AC-1 that you consider all pairs of variables but now you're going to maintain a queue essentially. And then we're going to make calls to revise. We get (xi, xj) out, so I'll use this notation that you're just dequeuing the queue and taking out the first element from that, which is equivalent to saying that you're removing it from the queue and now you're going to handle this.

Then I will call revise with domain i, domain j and Rij again. So again I will end up making 2*e calls to revise in the first cycle, exactly like we did earlier, each with a cost of k^2 essentially but now we do the following only if something has changed. What can change here? Di can change here. If Di has changed, then add those variables which are related to Xi to the queue. That's all we will do. In this example everything is related to everything else so it doesn't sound so interesting but supposing it was some less connected graph, there were many variables and there were fewer connections then if I change the domain Y, I will only be worried about those variables which are related to Y and I will put them in a queue essentially. So that's the basic idea behind this thing.

So if Di has changed, then for each Xk that is related to Xi, I must enqueue that variable, where $k \neq j$ and $k \neq i$. The variable Xi or domain of Di has changed, then I just want to make sure that all related variables are still consistent. So basically I put Xk, Xi in the queue and so at some point in due course of time that will also be looked at again. So again the revise to Xk, Xi will be made essentially.

So notice that I didn't make two specific calls to revise because the enqueue was done twice. So we did enqueue Xi, Xj and Xj, Xi. They will eventually come out one by one in the queue and each will get revised essentially and every time you revise a variable, then we look for everything that's connected to that and make a call to revise that. We put this in a loop as before till the queue becomes empty.

Now we want to talk about complexity of AC-3. Is it better than AC-1? What does the intuition say? It is better because you're only adding to the queue those variables which are related to a domain which has changed where as in AC-1 you do everything all over again. For all pairs of variables, do everything all over again.

We have now n variables and only one value has changed, DY in this case, then we'll only take those variables which are connected to Y and enqueue them. So we'll look at only a small subset of variables. Then if there is a change we'll look at another small subset and so on and so forth.

So what's the complexity of this? Again you can see that revise has the same complexity, $O(k^2)$. The first time we will do 2*e enqueues where e is the number of edges. So for both sides we will add one pair to the queue. The question we want to ask is what is the worst case complexity of this algorithm? Which means how many times can you add a pair of variables to the queue? Remember we are dealing at the granularity of the variable level. In fact the next algorithm that we will see, we will look at the granularity at the value level essentially. So if a value has changed, is that value changing another value? Now that's a more sophisticated algorithm which requires more sophisticated data structures.

We will see that later but here we are talking about a set of values. So how many times can each pair be enqueued? At the most k times, right? Because you know only k times that value will get changed. And there are e edges. So this whole thing will be done at most ek times. O(ek) times because there are e edges and k values and you're deleting values one at a time.

So what's happening here? You can see we are going to be following the propagation. So let's write the complexity for the previous example. Basically it's a product of these two things. So the complexity is $O(ek^3)$ essentially. Let me go back to the previous example which I had drawn here. When I do X with respect to Y, nothing will change. When I do Y with respect to X nothing will change. When I do Z with respect to Y nothing will change. And when I do Y with respect to Z, a value will get deleted. Then when I do Z with respect to X nothing will change. When I do X with respect to Z then one value will change.

So in the first cycle you can see that there are two values which have changed. And of these two values, if you see the graph, one is in X and the other one is in Y and the two elements at the two end points have got deleted in this first cycle. In the second cycle, the next two end points will get deleted. In the third cycle the third two end points and so on and so forth. So we will do it essentially ek times in the worst case because e is the number of edges in this. So the worst case of course happens when everything gets deleted from the corresponding domains.

Okay so let's just do a recap of what we have done. We looked at the algorithm AC-1which is a brute force algorithm which says revise every variable with respect to every other variable and if any domain has changed then do this whole process all over again – every variable against every other variable. Now that doesn't make sense because you know you're doing too much extra work.

So AC-3 says that initially you revise every variable with every other variable which is done by the two enqueue statements that we have but subsequently, only if a domain has changed, you look at the connected variables for that particular variable. So we are looking at the granularity of the variable level essentially.

Let me just give a small example here. I have X and I have Y. Now if I do AC-3, the only thing that will happen in the end of the first cycle is that the one coloured value will get deleted but you can see that deleting this value has not changed the consistency of Y with respect to X. That's what AC-3 does. AC-3 says if I'm changing the domain of X, then call revise Y with respect to X again. If Di has changed, then for each Xk that is related to Xi, put that in the queue and basically call a revise all over again.

Now in this small example on the bottom left you can see that after I've changed the domain of X, I've deleted one value and the variable Y is still consistent with respect to X. Assuming that you know there may have been another value which I had deleted but as a result of

deletion of this particular value, the one that is shaded here, the consistency of Y with respect to X is not changing. AC-3 will still investigate this. It'll still make a call for revise Y with respect to X which means for every value of Y it'll look for a value of X. This is the kind of work which it does because it's looking at the level of a variable.

In the next class, we'll look at the algorithm called AC-4 which will look at the level of values which will say is deleting this particular value changing something in some other variable? And then we'll try to follow the link. So we'll meet in the next class and look at algorithm AC-4 which requires a lot more of book keeping. So we need some nice data structures to handle that. Here we only needed a queue. So see you in the next class.

(Refer Slide Time : 29.46)