Constraint Satisfaction Problems

Prof. Deepak Khemani

Department of Computer Science

Indian Institute of Technology – Madras

Module – 3, Lecture – 04

We are looking at constraint propagation and in particular we are looking at arc consistency. We want to now look at the algorithm called AC-4 which keeps data at the value level. For every value a of every variable X, for every variable Y it is related to, it maintains a counter. We will call this a counter and it'll be indexed by the variable name X, the value that we are concerned about which is a and a related variable Y. When we say related we mean that it is a constraint. There is an edge between X and Y essentially. It maintains a counter XaY which is the count of support a gets from Y.

That's the basic idea behind algorithm AC-4. You have this structure called counter and you have one number for every value in every domain in the network. So you can see that the size of the structure is going to become quite large. So for every variable X, for every value a, you maintain a count, for each other variable, of how many supports it is getting from that variable.

The idea is that if the count becomes 0, then we have to delete a from D(X). This is propagation happening. Decrement the counters of all values in all variables supported by a to which a is one of the supports. That's the basic principle. That's the basic idea behind the algorithm AC-4 that you maintain a counter for every value and keep track of how many supports from each other variable it has. If from any variable, it's support becomes zero which means there is no value in the other variable which is related to this value a then you've to delete a. That's my definition of arc consistency. That's what revise would have done but now we delete a and we decrement the counts of all those values in all other variables that a was contributing to. So the next question is, which ones? How do you do this all values in all variables essentially?

So this idea is straight forward. You maintain another data structure. Maintain a support list. We will call this list S like most people do. S is indexed on the variable and the value. It's a

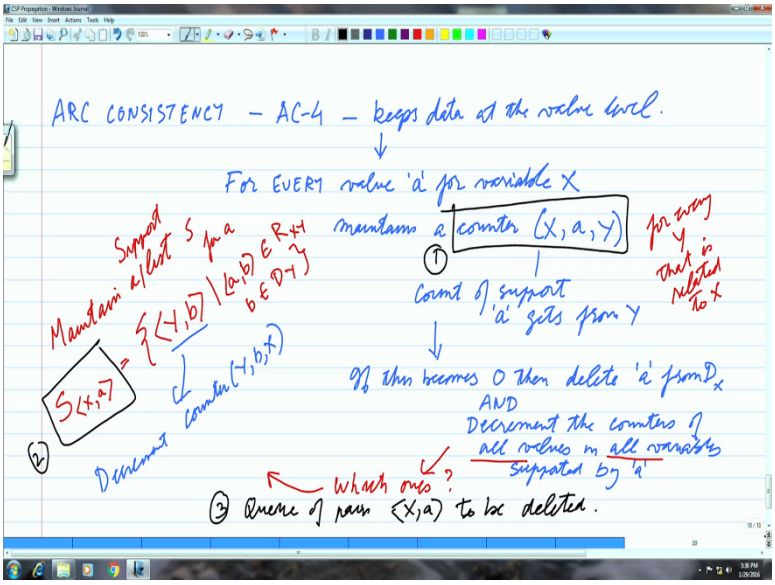
list of all those values that are related to a. So it's a list of all variable, value pairs such that <a,b> ϵ Rxy and b ϵ D(Y).

So these are two principle data structures that you want to maintain. One is a counter for every value that is there. The counter will tell you whether the value is supported or not and the counter is for each value, for each variable it is related to. So if variable X is related to3 other variables, then any value from X will have 3 counters. If any of them becomes zero, we've to remove a from that and then we have to worry about what a was supporting. So we have this structure called support. What was a supporting? It's support (X,a) and it's a list of all values in all variables whose value it is supporting.

Once we know this, what do you do? For each of these values, you decrement the counter YbX. So you know which is the variable because the support array is giving you a pair, (Y, b). It's giving you all such values. For each such value you say that the support it was getting from X has been decremented essentially. And then of course the rest of the algorithm is basically built around this that you keep track of counters. As soon as a counter becomes zero you delete it. You don't delete it immediately. You put it in a queue in which they'll get deleted because more than one counter could become zero at the same time. You essentially process elements one by one.

So what are data structures that we're using? One is the counter, one is the list and the third one is a queue of pairs of the kind (X, a) which are to be deleted. And when we delete it essentially we'll prune the domain and we will go and do the decrementing process, looking at the support S. Let's now try to describe the algorithm. We'll do it at a relatively high level and you can fill in the details.

(Refer Slide Time: 9.56)



The first phase is to initialize. Before you even start doing anything, you have to build these structures. You've to build these counters for every value because every value is related to more than one variable and you have to build the support structure S for all edges, XY, for each value a of X and likewise for the other variable.

You count the support from Y and when you're looking at Y you count the support from X. Suppose you want to set up (X, a). Just to emphasize, here we get values from Y essentially. So whichever elements a is related to, we get a pair. So out of (Y, b), (Y, c) and (Y, d); for whichever a is related to we'll get the counter and this is only for one edge. Likewise when we do for every edge we will get the whole picture.

What is the complexity of this? When you say all edges, this contributes a factor of e and then you're counting the support from Y, this will give you a factor of k^2. So it gives us ek^2. So simply setting up the data structure is order ek^2 essentially. And that's also the space required for S. So not only you need ek^2 time to set up the initial structures of counter and support, you also need space because for every pair of related values there are two entries. If a in X is related to b in Y, there is one entry in S, (X, a), which is the value (Y, b) and correspondingly there is another entry in S of (Y, b) which is the value (X, a) essentially. So there are all these ek^2 entries. So it requires a space of ek^2 entries essentially.

After this has been done, if any counter XaY becomes zero, there may be values which have no support. There may be a value in X which has no support from the variable Y so it's counter will be zero. We simply add it to the queue. Let's call the queue as list. It's a list but it basically doesn't matter in what order you process the pairs. So you can call it a list.

This is the initialization phase. For every value, you have a counter from every other variable and you know which values don't have any support so you know which values have to be deleted. That will happen in the propagation phase.

The second phase of the algorithm is propagation which we've already discussed. What it says is, for each <X,a> in list, first you delete a from its domain and then for each <Y,b> ϵ S(<X,a>), decrement the counter value for that variable. So, for the pair <Y,b> we got from the support list for <X,a>, it basically says that there is an edge from a in X to b in Y. This is all that it maintains. So if we've to delete a, the support Y was getting from the variable X has become less so decrement the counter by one and if counter becomes zero, then add <Y,b> to list.

Try to imagine the matching graph and see what's happening. You have some value a in X and there is another value b in Y and there is an edge between them. The edge is basically what the array S stores. It says a and b is an edge. It stores it for a as well as for b. It indexes on both the values. If a had an edge from c from some variable Z and you deleted c from Z, then looking at the array S for <Z,c>, it would have told us that <X,a> was related to <Z,c> so you'll decrement the counter for <X,a> and if this counter becomes zero, then we will put it in the list or queue and at some point in the future we will delete this a from the domain of X. Then because the S structure for X told us that a was related to b, you would be following that link to b and decrementing the counter for b. That's what the propagation phase is doing essentially – decrementing the counter.

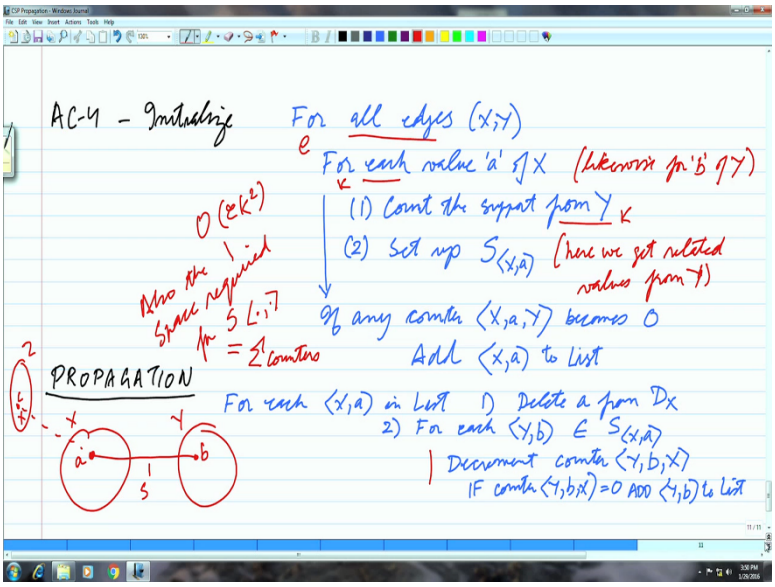
Essentially you're following the edges in the matching diagram as opposed to AC-3 where you were following the edges in the constraint graph. So if there is an edge and one end of the edge vanishes then you decrement the counter at the other end of the edge. And if the counter becomes zero you delete that.

What is the complexity of this propagation part? What is the worst case complexity? Obviously, you can delete only one value at a time. If you look at these S vectors, you may end up deleting whatever is stored in them. The space required for S is actually equal to the sum of all counters. If you just think about this a little bit because each counter tells you how many supports it has from another variable. So if you look at every counter from all other variables and count the supports you know how many elements are there in S. In the worst case you will delete all of them one by one. In the worst case, in every cycle of propagation, you will delete only one element. So you may require ek^2 cycles, which is the size of S

essentially. So you can see that the overall complexity of this is ek^2, which is better than AC-3 which was ek^3.

It turns out that in practice people have found that AC-3 often performs better than AC-4 even though AC-4 has a worst case complexity which is smaller than AC-3. The reason for this is that the best case complexity of AC-4 is also quite high. Simply setting up the initialization phase and setting up all the data structures requires ek^2 time which is not the case in AC-3.

(Refer Slide Time: 23.48)



So here is what we have seen about arc consistency. We have looked at three algorithms AC-1, AC-3 and AC-4, progressively looking at finer and finer granularity level and giving better and better worst case complexity measures essentially but the next question we want to ask is what does arc consistency imply? How does it help to make a network arc consistent? For the moment we're looking at only binary constraint networks but we can extend the idea of arc consistency to higher networks.

So the simplest question you might ask is does an arc consistent network, and I will just restrict myself to BCN for the moment, always have a solution? If you remember, the reason we had got into consistency is because we want to come down on search space but does it tell us anything more than that?

Unfortunately arc consistency for binary constraint networks is not enough to tell you whether there is a solution or not and there is a very simple example which illustrates this. If

you have a map colouring problem with red and blue as the two allowed colours three touching countries, then if I draw the edges you can see that every value in every domain has a value in the related domain.

You see every value has got a value in the related domain. So this means it is arc consistent but does it have a solution? If you look at this problem you'll see that there is no solution. You have two colours and three touching countries. You can't find a solution to this. We will see as we go along that the amount of consistency that you enforce will sometimes be enough to tell you whether there is a solution or not but unfortunately arc consistency is not enough in this example to say that there is a solution. What you need is the next level consistency which we would call as three consistency. Arc consistency was two consistency but if we have three consistency, which is also called path consistency, we will see that path consistency will tell you whether solution to the map colouring problem exists or not. Or at least to this network.

There is also a notion of generalized arc consistency which says that if the network is more than binary, if there are ternary constraints or higher order constraints do we have a notion of an arc consistency? There is one notion but I will describe it sometime later as we go along. So in the next class when we meet we will look at the notion of path consistency which is the next level of consistency.

(Refer Slide Time: 27.16)