Constraint Satisfaction Problems

Prof. Deepak Khemani

Department of Computer Science

Indian Institute of Technology – Madras

Module – 3, Lecture – 07

We are looking at path consistency and we saw that the brute force algorithm similar to AC-1, PC-1 was computationally expensive because it did a lot of unnecessary work. We moved from AC-1 to AC-3 and what we did in AC-3 was to propagate change. In AC-3, whenever a domain was pruned you saw if a related domain was affected. We'll do something similar in PC-2. I'll just discuss this at a high level. We'll maintain a queue of all revise operations that we need to call and only add to that queue if we have made some change. In this case the change will be deleting a pair from some relation Rxy.

Essentially, we will maintain a queue in which we will put three variables, say i, j and k and the idea is that I will call Revise((Xi, Xj), Xk) for all possible triples of variables. Keep in mind that this pair will be revised again with a third variable. That's how the revise call will be made. Revise-3((Xi, Xj), Xk). All such triples are added to queue. Remember that there are n cube number of triples essentially.

After Rij has changed, i.e., after pairs are pruned from the relation Rij, we need to add (l, i, j) and (l, j, i) for every l. What am I trying to say here? It's that if I have, let's say Xi, Xj and Xl, and I'm deleting the edge between Xi and Xj, then I must check for every other variable l whether this edge is three consistent Xi and Xj, that is, (l, j, i) and likewise I've to check for (l, i, j) because remember that path consistency says that every edge must be extendable to a triangle. l, i and j are indices of variable names. We are not talking about values. I'm just illustrating it by a matching diagram but the algorithm works, like AC-3, at a variable level. If a variable has changed in AC-3 you just call revise with a related variable. In this algorithm, if a relation has changed you just call revise-3 with every other variable essentially.

Now it may be the case that I have a situation where one edge of the triangle is deleted. I had a triangle but now the triangle is broken. I will need to worry about whether to keep the other two edges or not. I could have also had a situation like this where there's another triangle. It

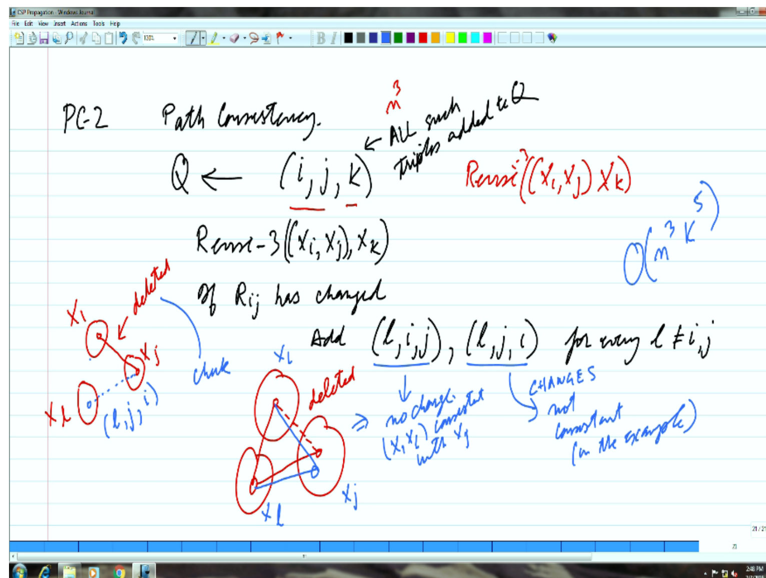
needn't be with a different variable. So if this is the situation then in the subsequent calls, in this example there will have no change which means (Xi, Xl) is consistent with Xj because there was another triangle in which (Xi, Xj) was participating in.

I must emphasize in the example. If a relation Rij had changed, then I need to call (Xl, Xi) once and (Xl, Xj) once essentially and in this example, we can see that one of those calls would have resulted in pruning of one relation but not the other relation but we are not looking at the matching diagram. Algorithm PC-2 doesn't look at the matching diagram. It is like AC-3. If a relation has changed, simply call it with other things. So, as you can see when we call revise with (l, i, j) there will be no change so nothing will be added to the queue but when we call (l, j, i) there would be a change in the edge between Xl and Xj and therefore we would have to add some more things to the queue. Things are only added to the queue when a relation changes essentially. So, this is just to illustrate the fact that in one case there is no change and in another case there is change and therefore this in turn will propagate its value to the next one essentially.

Let's discuss the complexity of this. The basic setting up operation will be $O(n^3k^3)$ because when we are forming the queue, we are making n^3 calls to revise because there are n^3 variables and each call costs k^3. That much is there but how much more will be added to that? If you look at that then you can argue that every edge will be added to the queue k^2 times at most because every pair of variables has k^2 edges between them and if you delete edges one by one then you will add it k^2 times into the queue. So, the complexity of this is $O(n^3k^5)$, which is better than $O(n^5k^5)$ essentially.

There is a version of path consistency algorithm which is analogous to AC-4. If you remember in AC-4 the propagation was done at the value level. There are implementations which exist but we will not discuss it here because it's too cumbersome to discuss that. I hope you get the general idea.

(Refer Slide Time: 11.11)



Let’s generalize i-consistency and one thing that I want to emphasize is that constraint propagation is a local operation. What I mean by that is that you only look at part of the problem and do something. It’s like making a logical inference. While making a logic inference you simply want to see if the antecedents are true and then you assert the consequent. You don’t look at the whole database to say what else is true. Constraint propagation is similar in nature that you just look at a subset of variables.

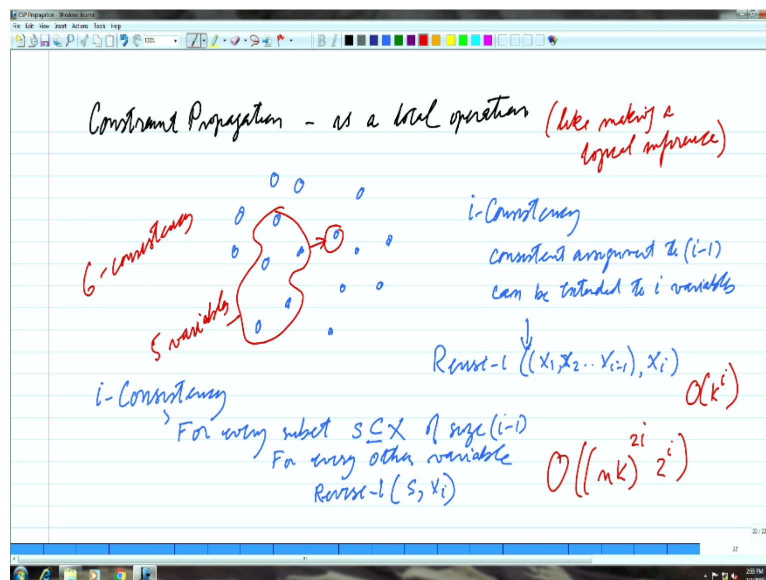
For example, let’s assume I have many variables. I’m drawing the nodes of my constraint graph and there would be some edges. In any case the graph itself doesn’t play a role from path consistency onwards because we don’t look at the edges when we talk about the algorithm. When we talked about path consistency, we said for all pairs of variables make it three consistent with every other variable. So, the idea of high consistency is that for all tuples of i-1 variables make it consistent with the ith variable essentially. That’s the notion of i consistency and enforcing i consistency would require a revise-i operation with appropriate set of arguments. You can say something like this – Revise$((X_1, X_2, X_{i-1}), X_i)$. It’s kind of extending the definition of revise three to Xi essentially which means that you’re only looking at a subset of the variables and then changing that subset itself.

So, if I’m talking about, for example, six consistency, then I would look at every set of five variables and I will ask, can I extend it to a sixth variable? You remember we’re talking at the level of a constraint graph. In that sense that’s what I meant by saying that constraint propagation is a local operation. That you look at subsets of things and make some inferences and change things but it has a global impact. Your whole network can become i-consistent if

you do this for all combinations of, in this example, five variables but now all combinations of five variables are quite a few. That's why it tends to become more and more expensive.

We had kind of made a guess about the complexity. We will leave it at that that the revise operation is $O(k^i)$. The algorithm for i-consistency I will not write in detail but the key step is, for every subset S of size i-1 of a set of variables, and for every other variable you call revise. Of course, there are the conditions that how many times should you do this loop. So, there would be some looping here but if you're doing something like PC-2 at some point you may form a queue but the brute force algorithm has complexity $O((nk)^{2i}\, 2^i)$. This is what deters us from trying to do full consistency. It tends to get more and more complex essentially.

(Refer Slide Time: 17.22)



So, we saw that for BCNs strong path consistency implies solution exists. Let's very briefly dwell on the words strong again. I have this network of the map colouring example again. This is, as we had observed, AC but not PC. This is just a three countries two colours problem. You cannot colour three touching countries with two colours and this is just an abstraction of that. If you look carefully there is no triangle in this. It looks like there are triangles but there is no triangle inside this essentially. So, this network is not path consistent.

If we made it path consistent, so let's say we do PC-1, then what do we get? All the edges vanish because path consistency says that if any edge cannot be extended to a triangle delete that edge and eventually all six edges will get deleted. So, we'll end up in a network which

has no edges. Trivially it is path consistent because there are no edges so you can't extend it; but it is not arc consistent.

Now if you do, let's say AC-1 to this, we will get empty domains and an empty domain means no solution. In fact, whenever you write an algorithm for solving a CSP, at any point, even if a single domain becomes empty, you can simply terminate and say that no solution exists because the very fact that a domain is empty simply means that you cannot get a value for that variable and if you cannot get a value for some variable you cannot get a solution for that problem.

So, this network that we started with was not path consistent. When we made it path consistent we found that it was PC but not AC. This makes it not strong. So, it's path consistent but not strongly path consistent. Now if you try to make it arc consistent then all the values from the domain should have been gone and that means it's not even node consistent. I mean the node itself don't have consistent values to assign.

In general, if a network is strongly n-consistent then it has a solution. n-consistency says that if you have a partial solution of n-1 variables, you can extend it to the nth variable but to have a partial solution of n-1 variables it should be n-1 consistent essentially which means that you can extend everything. So, you need this strongly consistent property for you to say that a solution exists and making a network strongly consistent is computationally expensive. We have not tried to write the algorithms for doing that but this is the general approach to make it – you first make it PC then you make it AC then you make it node consistent and so on because every time you are making it i-consistent you are inducing a relation of i-1. Then you should be sure that at that level it is consistent. So, you have to do i-1 consistency. Then you have to do i-2 consistency and so on. This is called global consistency. It's a very expensive process and we do not normally try to do that.

So we will try to look at combinations of doing other things. First we will try to see whether we can exploit the order of variables to do less consistency enforcement and then later on we will even try to see whether we can modify the search algorithm to combine consistency with search. You look for a variable and then you look ahead to see whether you will have values for future variables. It's a whole area of quite exciting algorithms that we will be looking at. I will end here with an exercise.

You have a given network R in which there are three variables – {x, y, z}. Each domain has just two values – {0, 1} and there is only one constraint between the three variables Rxyz

which allows only one combination – {0, 0, 0}. The question is, is this network path consistent? Is it strongly path consistent? I want you to look at this very simple network. There is only one relation of arity three in this. It says that you can only assign value zero to variables. That's the only constraint you have. So is this path consistent and is it strongly path consistent? And if you want to make it consistent, what will the new R' look like once you make it consistent? I will leave this as a small exercise.

In the next class, we will move towards the idea of directional consistency in which we exploit a direction. We order the constraint graph in the order in which we are going to process things and only look at consistency in one direction.

(Refer Slide Time: 25.39)