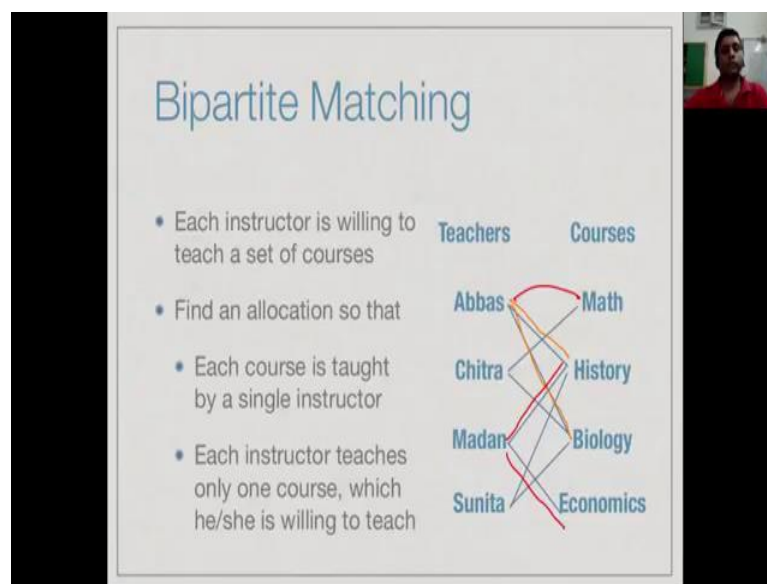


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 08
Module - 05
Lecture - 54
Reductions

We have seen that we can use linear programming as a technology to solve a number of problems and formally this involves what we call a reduction.

(Refer Slide Time: 00:12)

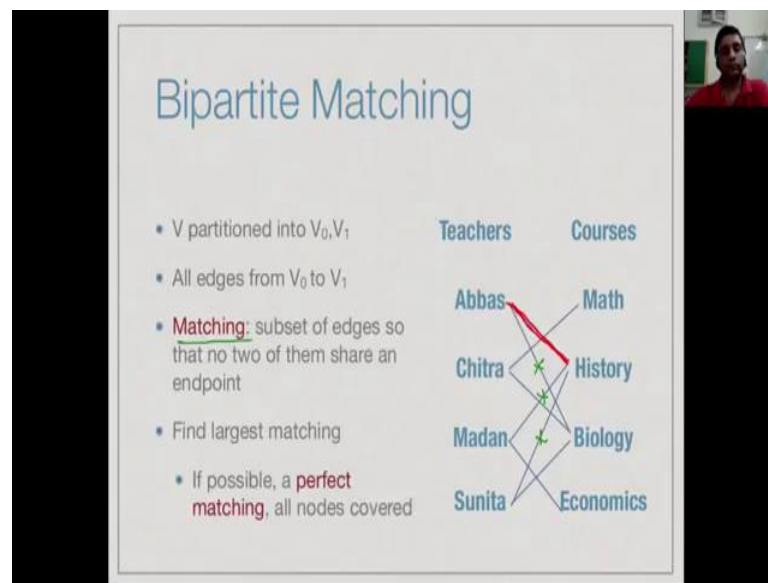


So, rather than reducing to linear programs, let us look at how we can actually reduce the network flows which we saw already could be reduced linear programs. So, here is a problem which seems to our nothing to do with flow. So, we have some course allocation to be done in a school, we have a collection of teachers and we have a collection of courses and each teacher as indicated the preference of which courses he or she is willing to teach.

So, here we have 4 teachers Abbas, Chitra, Madan and Sunita and we have four courses Math, History, Biology and Economics. So, Abbas for instances he is willing to teach History and Biology, so that is these two edges, so this indicates that Abbas is with teaching these courses. Similarly, for a since if we can look at Madan, Madan is happy to teach History or Economics. So, each teacher has indicated as a collection of courses that he or she is willing to teach.

Now, what we have to do is allocate these courses to the teachers. Obviously, each course is going to be taught by exactly one teacher, but also we would like that each teacher teaches only one course and that course should be something that the teacher is willing to teach. So, we would not like an allocation for example, which ask Abbas to teach Math, because he is indicated that he is not comfortable in teaching Math. So, we want to find such an allocation that each course is taught by a single instructor, each instructor teaches only one course and that is a course which he or she is willing to teach.

(Refer Slide Time: 01:42)



So, this is what is called as a matching problem, in particular it is called bipartite matching. So, what is a bipartite graph? Bipartite graph is which one in which the vertices come in two groups which we called v_0 and v_1 and all the edges go from v_0 to v_1 . So, there are no edges inside v_0 , so there are no edges inside v_1 . So, this is an example here, so this program to the teachers are v_0 and the courses are v_1 , now there are no edges between courses, there are no edges between teachers, all the edges are from a teacher to a course.

Now, what we have asked is to find an allocation. What will an allocation do? An allocation will pick an edge and say for instance that Abbas will teach History. And once Abbas is for been assigned History, then this means that this edge cannot be taken anymore, because Abbas cannot be assigned anything else, it also means that this edge cannot be taken, nobody else can get assigned this, this edge cannot be taken.

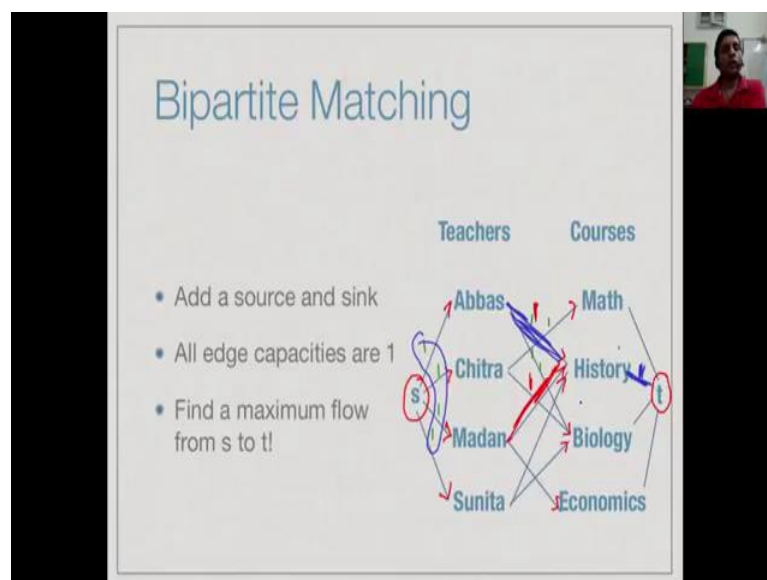
So, we want to find edges, such that no two of them share an end point. In this example,

if two of them share an end point on the left, it means that the same teacher is being asked to teach two courses. If that is two edges share an end point on the right, it means that the same course has been taught to two different teachers, either of it is which we want. So, you want a matching, we want a matching which is a subset of edges, so that no two of them share an end point.

And in particular, if we have an equal number of nodes on each side in a bipartite graph, then we can expect or hope that everything on the left is matched to something on the right. If we do not have an equal number of course, something has been left out. But each edge would match out two people, suppose we have a five teachers and four courses, then one teacher will not teach a course or we have four teachers and five courses, then some course will not be taught unless the teacher take two courses.

But you have four teachers and four courses, you could ask whether there is a way of matching them up, so that every teacher teaches a course and every course is taught by a teacher. This is what is called a perfect match. So, what was this going to do with network flows?

(Refer Slide Time: 03:48)



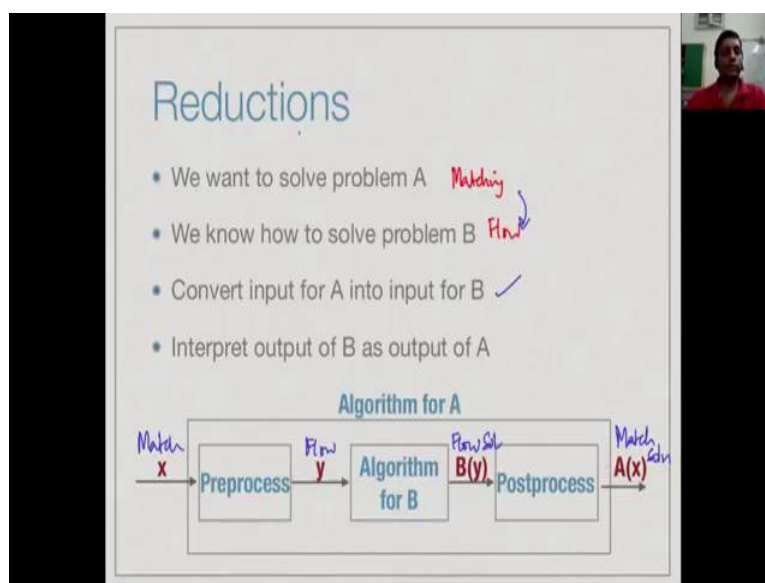
So, here is how we can answer this question using network flows. We add a spurious source node feeding into the teachers and we add a spurious target node, a sink node coming out of the courses. So, implicitly all are edges are now going from left to right, so we have the direction, so we are trying to flow something from left to right. Now, what we want is to assign capacity one to everything, so every edge in this has capacity

one.

So, supposing we find a maximum flow, so our maximum flow will select some subset of these teachers, hopeful all of them. Because it is a flow that flow will go into a teacher and out of a teacher, so I will select one of these edges, so this one. And therefore, now it will flow from this out here, now because the output can only take one, I can only flow 1 from History to the sink, it cannot allow another thing to coming from here.

So, if I had a flow 1 here and flow 1 here, then 2 would have to flow to the History, but I have got it. So, I putting a flow capacity of 1 on all the edges and I am sure that exactly one teacher and one course can be selected in the matching. And now, if you maximize this flow, it maximizes the number of teachers who are connected to their courses and therefore it maximizes the matching. So, we have managed to take a problem involving something very different involving graphs and modulate it using network flows.

(Refer Slide Time: 05:24)

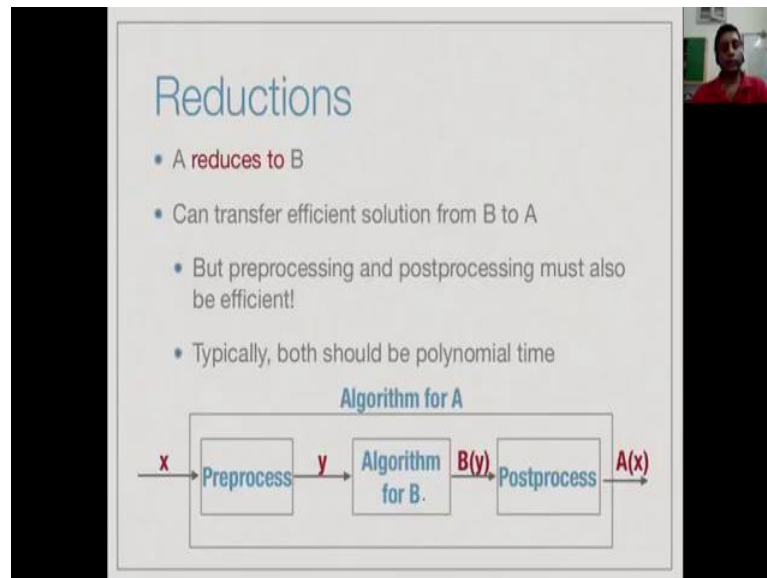


So, this is a general example of what we called a reduction. So, we want to solve a given problem, problem A, but we do not know how to solve it, but we do know how to solve another problem B. So, in our case this is matching and this is flow. So, what we do is we transform that problem from matching to flow. So, we take our matching problem and then we add a source, we add a sink, we add capacities convert it into a flow problem.

So, having converted it into flow problem, we solved it, so this is the diagram shown below, we have an algorithm for B. So, now, we have a matching input, we converted it into a flow input, this is a flow solution that comes out and now we look at the flow

solution wherever, we see a one that is our matching, whenever we see a zero that is not it our matching. So, from that we get a matching solution. So, we can involve the solution from the inner problem to get a solution to the outer problem. So, this is a reduction.

(Refer Slide Time: 06:25)



So, we say that A reduces to B. To solve A, I can convert it to B and solve B in state. And therefore, if I have an efficient solution for B and if this conversion process is efficient, then the process of setting up the problem for B and interpreting the answer are both efficient. Then, that is the efficiency of the algorithm for B composed with the efficiency of the pre processing and the post processing step will give us an efficient algorithm for A. So, without directly attacking A, I indirectly found an efficient algorithm for A by exploiting the existence of an algorithm for B.

(Refer Slide Time: 07:03)

The slide is titled "Reductions" in blue. It contains three bullet points:

- Bipartite matching reduces to max flow
- Max flow reduces to LP *f_e for each edge e*
- Number of variables, constraints is linear in the size of the graph

Below the text is a diagram labeled "Algorithm for A". It shows a sequence of steps: an input x (in red) enters a box labeled "Preprocess". An arrow labeled y (in red) leads from "Preprocess" to a box labeled "Algorithm for B". An arrow labeled $B(y)$ (in red) leads from "Algorithm for B" to a box labeled "Postprocess". Finally, an arrow labeled $A(x)$ (in red) exits the "Postprocess" box. The entire sequence from "Preprocess" to "Postprocess" is enclosed in a larger box labeled "Algorithm for A".

So, as we said before our example that we did in this lecture says that bipartite matching in this way reduces to match flow. Now, what we saw, last time it was a max flow itself reduces to linear program, but the key is that the pre processing and the post processing must be efficient. When we went from max flow to linear programming, what we did was we introduce one variable f_e for each edge e . So, therefore that is an efficient translation, because it produces the linear program whose size is comparable to the input graph of max flow.

Earlier, we had seen a example of network bandwidth, where we produce a linear program which require one variable per path in the network, that could not be an efficient reduction, because that would require an exponential amount of work to construct the linear program and to read the program. So, therefore, we have to be careful in order to if we had, only attention you solving the problem is one thing to convert a problem, but if we want to exploit the efficiency aspect, then the translation two and from the inner problem must also be efficient.

(Refer Slide Time: 08:13)

The slide is titled "Reductions" in blue. At the top, there are handwritten notes in green and red: "not eff A → B" and "not eff B ← A", and "efficient ← efficient". Below the title, there are three bullet points:

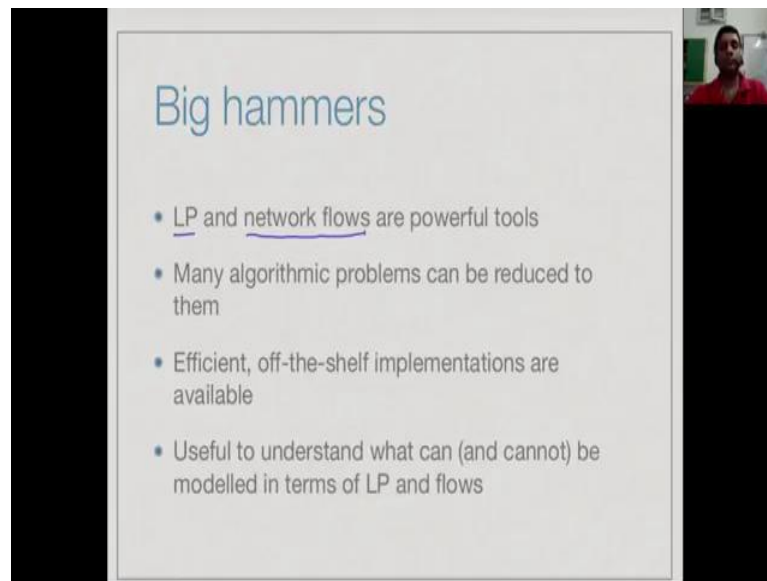
- Reverse interpretation is also useful
- If A is known to be intractable and A reduces to B, then B must also be intractable
- Otherwise, solution for B will yield solution for A

Below the bullet points is a flowchart titled "Algorithm for A". The flowchart shows an input x entering a box labeled "Preprocess", which outputs y to a box labeled "Algorithm for B". The "Algorithm for B" box outputs $B(y)$ to a box labeled "Postprocess", which finally outputs $A(x)$.

So, we will see that another interpretation of reduction also useful for us. So, what we have seen that is that if A reduces to B, then B is efficient then this implies that A is also efficient, because I can use the solution for B also a solution for A. But supposing I have that this is not efficient or not known to be efficient, if I have some reason to suspect that there is no efficient way to do A, then if can show that A can be reduced to B, then this also means that B cannot be efficient.

Because it could be done efficiently, we know that we can do way efficiently, but we have a reason to believe that A cannot be done efficiently. So, reduction can be used depending on the way in which we exploited either to transfer a positive result from B to A or a negative result from A to B, that is A is not known to be efficient, B also is not known to be efficient.

(Refer Slide Time: 09:10)



Big hammers

- LP and network flows are powerful tools
- Many algorithmic problems can be reduced to them
- Efficient, off-the-shelf implementations are available
- Useful to understand what can (and cannot) be modelled in terms of LP and flows

So, in this last couple of lectures what we have seen are two, what I would call Big hammers. So, we have linear programming and network flows. We actually express network flows in terms of linear program, but network flows independently are a very powerful modeling formalization. So, many algorithmic problems can actually be reduced one of these two, either you can set up variables in a set of linear program or you can take a graph erratic problem and modulate it using flows.

And the nice thing about this is that linear programming and network flows are both very standard problems for which people have interested, have lot of time to write general purpose tools which solve this problem. So, you can set up an arbitrary linear program, arbitrary network flow give it to the tool, then the tool will give you an answer. So, you have efficient of the self implementations. So, in many practical situations rather than try and solve an algorithmic problem directly, it does often make sense to think of whether it can be modeled as a linear program or a network flow and then use an off-the-shelf package to solve it.

But of course, you have to be able to express the problem in terms of one of these things. So, in order to understand whether you can do so or not, you need to spend some time analyzing what can and cannot be express the linear programs. In linear program, the crucial thing is that the constraints must be linear functions. Sometimes, your constraints which referred to the product to two variables, this is not a linear function.

Similarly, in network flows you have to make sure that you can actually modulate as a

source to target flow, it cannot be something going around and around in the network. So, within these constraints; however, both of these are extremely useful generic problems for which efficient solutions do exist and which cannot be exploited in a large number of situations to solve the given problem at hand.