Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

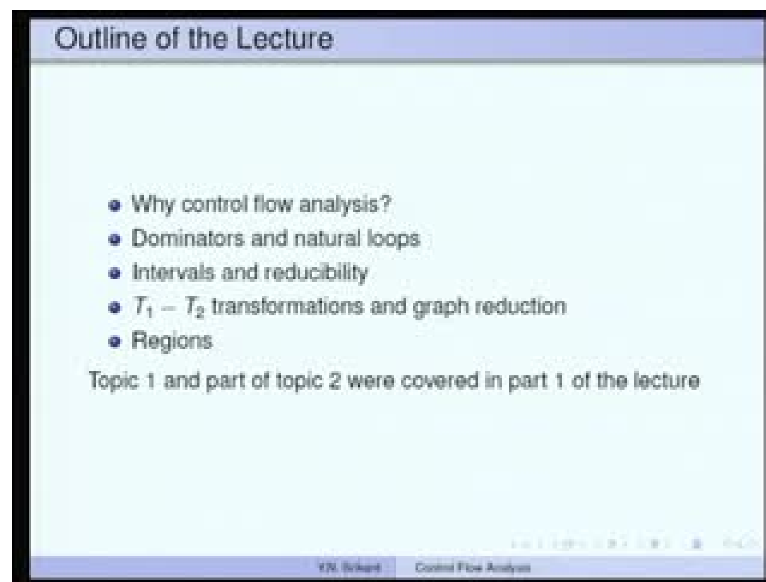
Indian Institute of Science, Bangalore

Module No. # 09

Lecture No. # 15

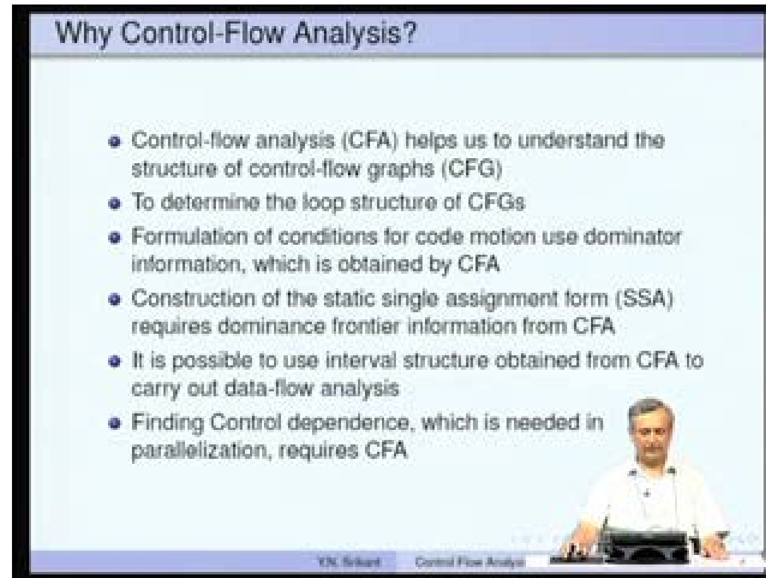
Control Flow Analysis-Part2

(Refer Slide Time: 00:22)



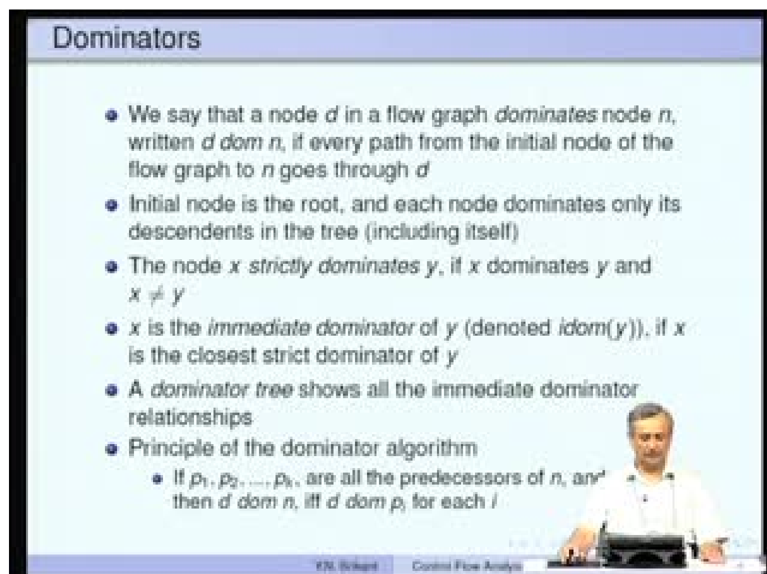
Welcome to part 2 of the lecture on control flow analysis. In the last lecture, we covered dominators and little bit about natural loops. We will continue the part about natural loops and then go on to intervals, reducibility, etcetera.

(Refer Slide Time: 00:41)



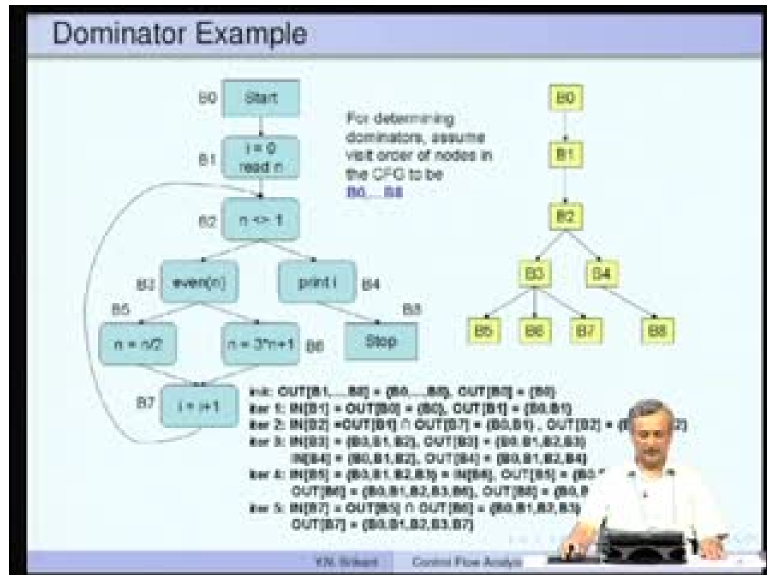
To do a quick recap: Control flow analysis is a essential to discover the loop structure of control flow graphs; it also provides us with dominator information which is needed almost everywhere - for example, to determine loops, to determine dominance frontier information. The control flow analysis also helps us to discover interval structure and so on. Control dependence, which is essential for parallelization, is also a byproduct of control flow analysis.

(Refer Slide Time: 01:20)



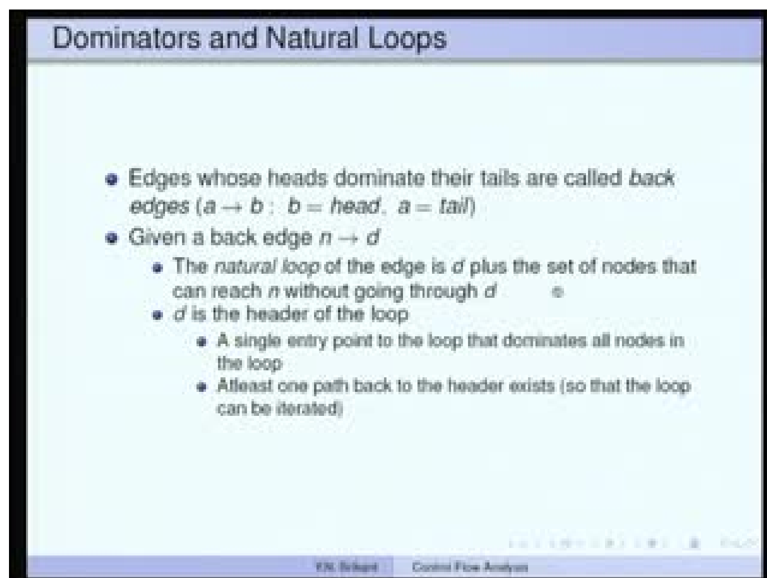
Quickly - dominators are very important. Let us do a quick recap of that - you can do that with this example itself.

(Refer Slide Time: 01:28)



If you look at this picture, let us say, B1 dominates B6 because every path starting from the initial node B0 to B6 goes through B1. That is the way the dominator information is also computed and is represented in the form of a dominator tree.

(Refer Slide Time: 01:54)

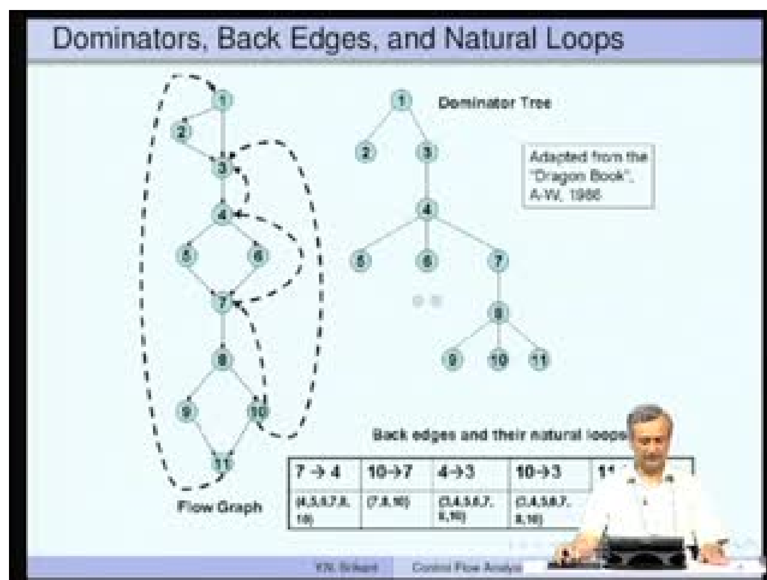


Now, what exactly is a natural loop? Once we discover dominator information, you find out those edges whose heads dominate tails. If a to b is an edge and b is called the head and a is called the tail; the arrow part is the head and other part is the tail.

If the head dominates the tail, then it is called as a back edge. First of all, we need to discover back edges, which is very easy. You just have the dominator information; look at all the edges and whenever heads dominate tails; that is called as a back edge.

What is a natural loop? A natural loop is defined with respect to a back edge. For example, the natural loop of an edge is the node d itself plus the set of nodes that can reach the tail n without going through d. You are not supposed to iterate through the loop which you can see physically, but you should be able to reach all other nodes in the loop. d is called as the header of the loop; it is a single entry point for the loop and that dominates all other nodes in the loop. At most, one path back to the header exists so that the loop can be iterated.

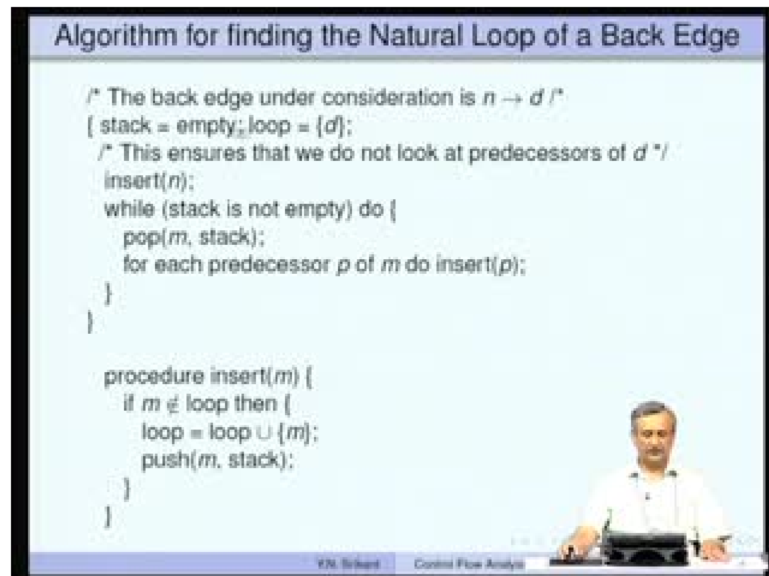
(Refer Slide Time: 03:31)



Here is an example. These are the back edges: 7 dominates 10, 4 dominates 7, 3 dominates 4, etcetera and the loop structure corresponding to the back edges is also here. Let us take a very straight forward edge; let us say 7 to 4 and now 4 is the head and 7 is the tail.

So, 4 dominates 5, 6, 7, 8 and 10; you can see that from the dominator tree 4, 5, 6, 7, 8, 10. These are all dominated by the node number 4 and then if you look at the reachability, you can also reach all these nodes from 4. All the requirements are satisfied. Let us look at the algorithm, which actually constructs these loops.

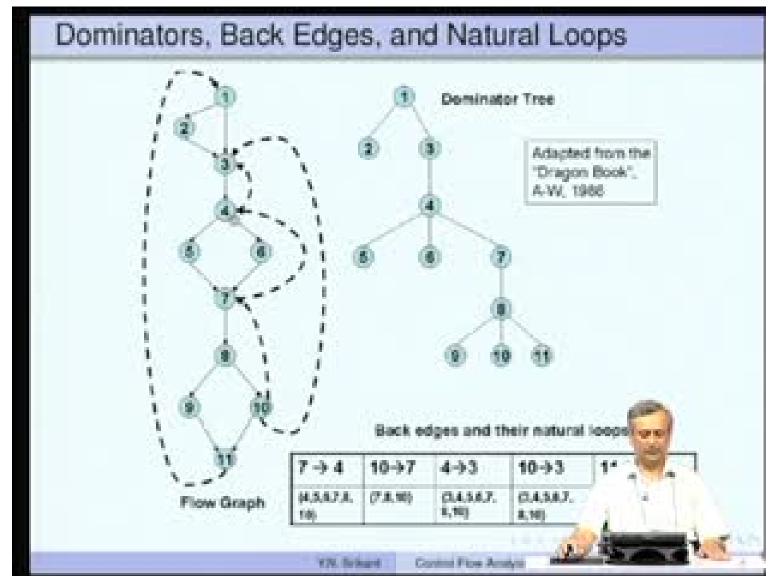
(Refer Slide Time: 04:33)



You are given the back edge under consideration as n to d and then this algorithm uses a stack. The stack is, to begin with, empty and the loop contains just the head and the reason why we put this head into the loop to begin with is to ensure that we do not look at predecessors of d anymore.

Otherwise, we are going out of the loop; we do not want to do that. The first statement is insert n. Let us look at the procedure insert which is right here. Procedure insert is very simple. If the node m, which is the parameter to insert, is not in the loop then add it to the loop and then push the node on to the stack. Once we do the insert on the tail node n, we start backward tracing from the node n. While stack is not empty, do pop the node from the stack and for each predecessor p of m, do insert p.

(Refer Slide Time: 05:44)



Let me explain this procedure with respect to this particular example. Let us take the same edge 7 to 4 which is a back edge. To begin with, d is inserted into the loop; 4 goes into the loop and then insert 7 is executed; 7 is inserted into the loop. 4 and 7 are now in the loop and 7 is also pushed on to the stack. Now, in the loop, we pop 7 and look at the predecessors of 7. (Refer Slide Time: 06:19) That is what this is saying; pop and then for each predecessor do insert.

The predecessors of 7 are many; 5 and 6 are two of them and then you also have 10. So, we have 3 of them. When we do an insert operation on 5, it goes on to the stack and also into the loop.

Then, we do insert operation on 6, which is also put into the loop and pushed on to the stack; then we go on to 10, insert it into the loop and push it on to the stack. 10 is the last node pushed; so, 10 is taken out. 10's predecessor is 8; so, 8 is pushed on to the stack. Then because of 8, when it is popped, 7 is pushed onto the stack and inserted into the loop and of course 6 will also be inserted in to the loop. We do not we go beyond 4; we just stop there.

Predecessors of 4 are not looked at. That is how we insert 4, 5, 6, 7, 8 and 10 and since we do not have any of the other nodes 9, 10, 11, etcetera as predecessors of the node; they are not inserted into the loop.

Let's check the edge 10 to 7. 7 and then 10 and then the predecessors of 10 which is just 1 node, that is, 8, because 7 is already inserted and 8's predecessor is 7. So, 7, 8, 10 is a loop on its own.

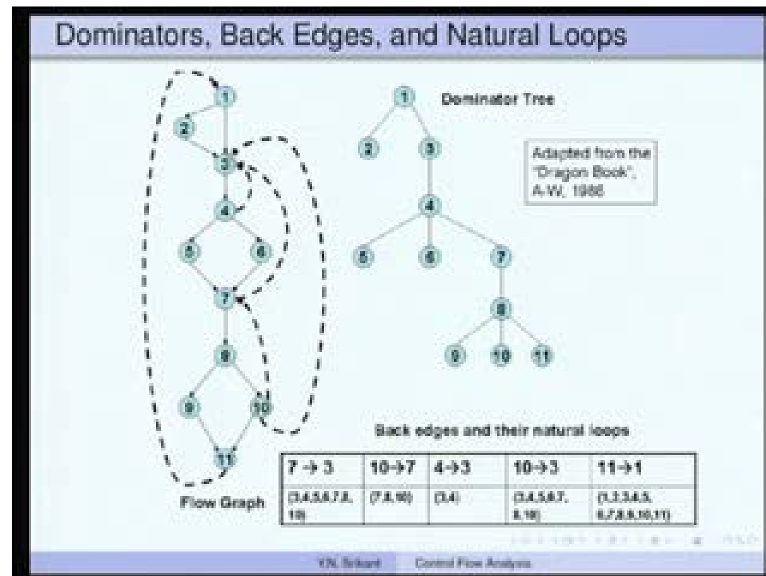
The surprising thing is about 4 to 3. Intuitively, we see that 3 to 4 is a small loop right here, but when we execute the algorithm on this particular back edge 4 to 3, there is a surprise in store for us. 3 and 4 are inserted into loop and 4 is pushed on to the stack. Then we start looking at the predecessors of; 4 that gives us 7 and 7 is inserted into the loop; the predecessors of 7 are 5 and 6; they are inserted into the loop. 10 is inserted into the loop and also on to the stack; then that gets us 8. So, we have 3, 4, 5, 6, 7, 8, 10 - all of them inserted into the loop and that is the entire loop structure as far as 4 to 3 is concerned.

Even though the result is surprising, if you look at it, there is an intuitive explanation. It simply says that without going to 3, we still have many paths for travelling to 3 after we reach 4. Let us say from 3 we go to 4, then we can go to 5, then 7, then 4 and then 3 or we can go to 4 ,5, 7, 8, 10 and then go to 7 and then to 4 and then to 3 etcetera. So, there are many of these possibilities. That is why it is even though it is counter-intuitive from the picture, this loop structure is correct.

The next one is 10 to 3. (Refer Slide Time: 09:43) That is this; this particular back edge. If you look at it, 3 and 10 are inserted into the loop and then starts the back tracing. So, 10 makes sure 8 and then 7, 5 and 6 and then 4, all these are inserted into the loop; so, that makes it 3, 4, 5, 6, 7, 8 and 10. This entire thing becomes another loop. You can see that 4 to 3 and 10 to 3 are identical - 3, 4, 5, 6, 7, 8, 10 and 3, 4, 5, 6, 7, 8, 10.

Even though physically we see 10 to 3 kind of nesting 4 to 3, the loop structure is really identical; both have exactly the same loop structure. Then, 11 to 1 of course, intuitively also it is correct. It encompasses the entire control flow graph itself. So, this is the way loop structures are determined using that algorithm.

(Refer Slide Time: 10:45)



Here is another example with a minor change. Here we had 7 to 4 and then 4 to 3. In this example, we have 7 to 3 and 4 to 3; let us see what happens. This so called minor change changes the loop structure of 4 to 3. From 4, you really have no other edge; 4 is not a predecessor of any other node except 3. So, 3 to 4 itself happens to be a loop and nothing else enters that particular loop, but 10 to 3 remains the same. 10 to 3 remains the same and it has all other edges. 7 to 3 is also the same. There is no difference between these. You see that 7 to 3 really nests 4 to 3 and then 10 to 3 nests 4 to 3 and so on. So this is the natural loop structure of control flow graphs.

(Refer Slide Time: 11:50)

Depth-First Numbering of Nodes in a CFG

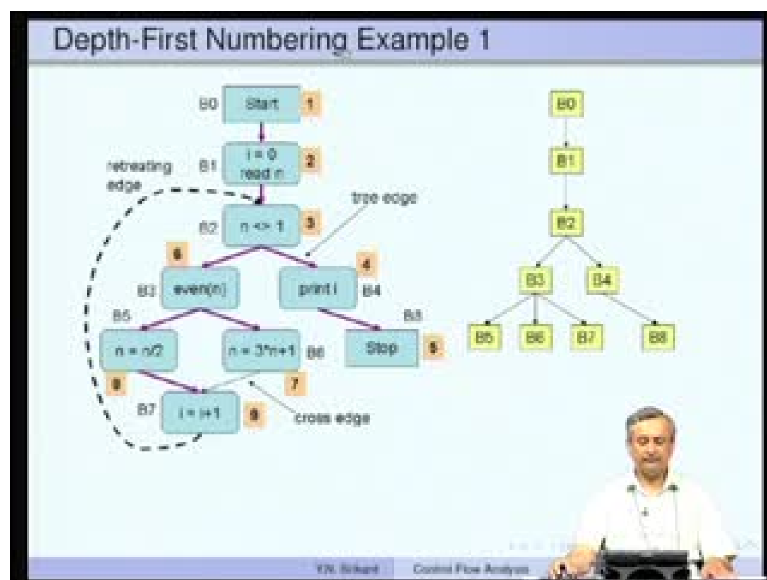
```
void dfs-num(int n) {
    mark node n "visited";
    for each node s adjacent to n do {
        if s is "unvisited" {
            add edge n → s to dfs tree T;
            dfs-num(s);
        }
    depth-first-num[n] = i ; i-- ;
}
// Main program
{  T = empty; mark all nodes of CFG as "unvisited";
   i = number of nodes of CFG;
   dfs-num(n0);// n0 is the entry node of the CFG
}
```

We must now look at a very important numbering scheme called depth first numbering. Well, we may wonder why we are looking at the depth first search and depth first numbering in this course on compiler design. The numbering here is slightly different from the numbering that we do in the depth first search. The visit order of depth first search is not exactly the same as dfs numbering, the depth first numbering that we study here. This numbering is used to actually define the depth of a loop and things of that kind; so, it is necessary to study this as well.

The depth first search is known to all of you. This is very simple and a very important searching strategy on graphs. Let us look at the main program. The depth first produces a spanning tree of graph. To begin with that spanning tree is empty and we mark all the nodes of CFG as unvisited; this is as usual as in depth first search. Then i is the number of nodes in the control flow graph and we call dfs-num on n_0 - n_0 is the entry node of the CFG.

So we start the depth first search; mark the node as visited and then for every node which is adjacent to this particular node n. If s is the adjacent node, if s is unvisited that is not yet visited, add the edge n to s to the depth first search tree T and then call dfs-num on s. Now is the numbering part; depth first num n equal to i and i minus minus that is equal to i minus 1. What this particular procedure is trying to do is number the node after the last visit to that particular node is completed. Let me explain this with an example

(Refer Slide Time: 14:04)

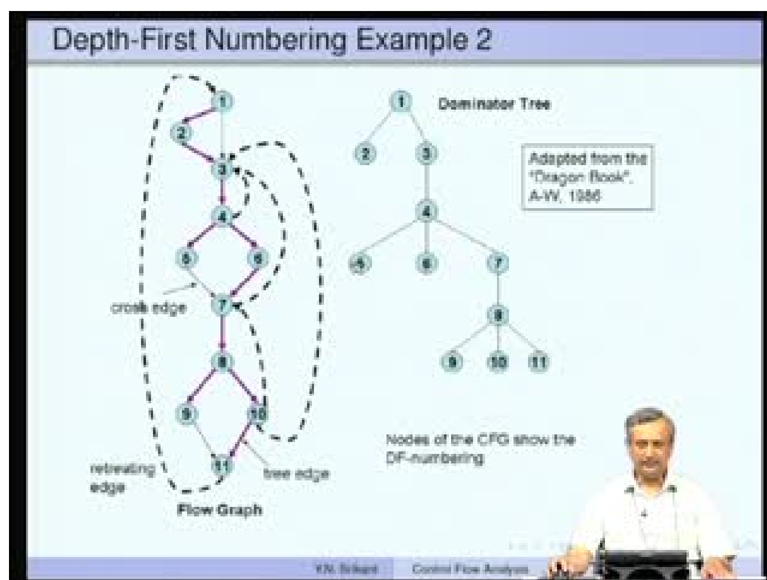


We start depth first search from B0. Depth first numbers are all given in this 1, 2, 3 etcetera; in the orange yellow are the depth first numbers. We start from B0; remember that the counter i was initialized to number of nodes. In this case, we have 9 nodes 1, 2, 3, 4, 5, 6 7 8 9. We have initialized the counter to 9. We start the depth first search from this point B0, then we visit B1, then we visit B2, then let us say we visit B3, then we go to B5 and then finally, to B7. This is one possible depth first search and at B7, we cannot go any further because there is an adjacent edge to B2, but B2 is already visited.

Now after this, we do not visit B7 again because it has no other neighbours that can be visited. The numbering for B7 will be 9; this is a last visit to B7. Then we back track, go to B5. Of course, B5 will not be visited again; that is the last visit. There are no other adjacent nodes for B5; so, it is given the number 8. Then we return to B3; there is one more neighbour which is not yet visited. We go to B6 and then from B6, there is nothing more to do because B7 is already visited.

So this is the last visit to B6; that is given the number 7. From 9 to 8 to 7, we return to B3; that is the last visit, it gets 6. Then we go to B2, there are more things to do. We visit B4 and then B8; nothing more from B8. That is given the next number 5. Then return to B4 gets number 4, this 3, this is 2 and this is 1. This is depth first numbering of this particular control flow graph.

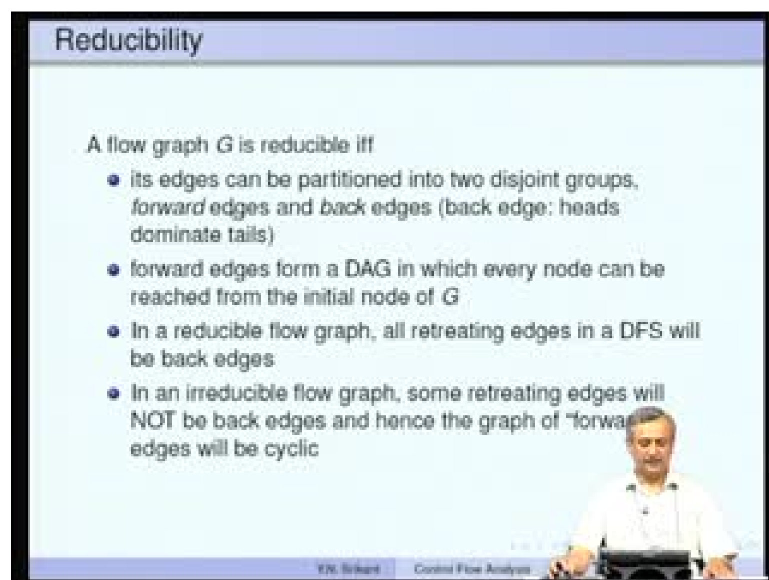
(Refer Slide Time: 16:19)



If you look at our running example, it so happens that the nodes are numbered in the df numbering sequence itself; we start from 1 then you go to 2, 3, 4, 5, 4, then 6 and then 7 and then 8 and then 10 and 11. Let us say that is the last node. You do not have any more nodes to visit from here. We start the numbering from here; so the number of nodes is 11 and this gets 11.

This will get 10 and then we go to 8 and go back to this. This gets 9, then 8, then 7, then 6, then we go back to 4 and then to 5, that gets 5 and 4, then 3, then 2 and then 1. So, this numbering is as it is depth first numbering of the control flow graph.

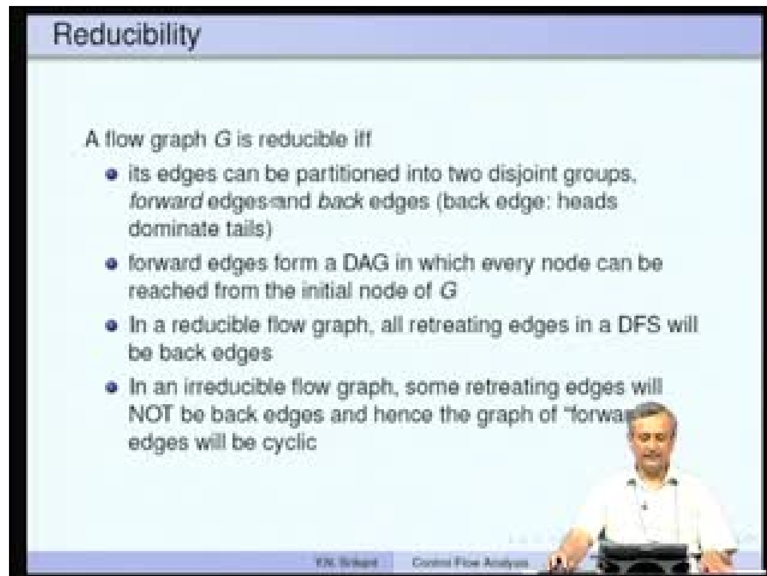
(Refer Slide Time: 17:12)



DFS numbering can be used to define certain quantities. We will see a little later. Now, we come to a very important concept called reducibility. (Refer Slide Time: 17:47) We saw that in the depth first search we really created a tree from the control flow graph and there were many other edges, for example, these edges which were actually not added to the tree. These edges we know already, they are all called back edges, but the general term that is used for such edges is called a retreating edge. What is a retreating edge? It goes from a particular node to its ancestor in the tree.

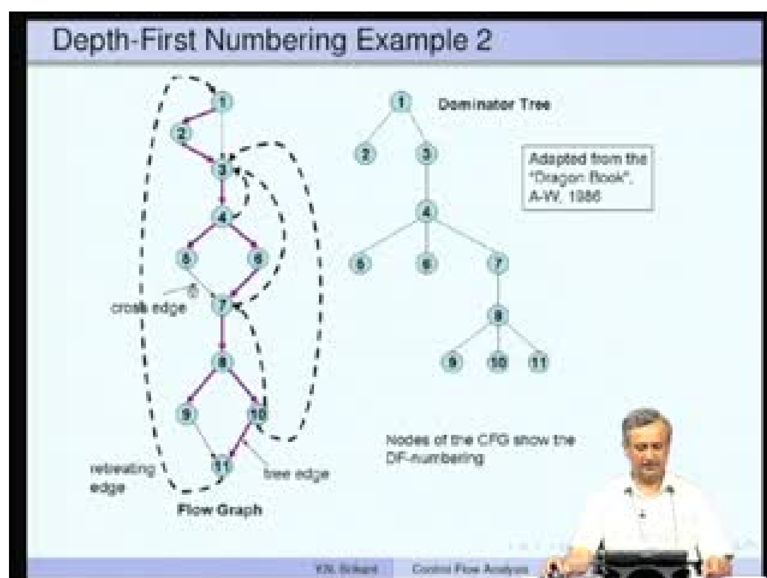
So, that is a retreating edge and then there are edges, which actually goes forward from a node, but if you add that particular edge it becomes not a tree, it violates the property of a tree. Such edges are called cross edges and then the edges within the tree are called tree edges. What is special about the back edge?

(Refer Slide Time: 18:42)



See that is the notion of reducibility. A flow graph G is reducible if and only if its edges can be partitioned into 2 disjoint groups: forward edges and back edges. So, back edges are the ones were the heads dominate tails.

(Refer Slide Time: 19:33)



Forward edges from a DAG in which every node can be reached from the initial node and in a reducible flow graph all retreating edges in a depth first search tree will be back edges. This is a very important property. We have forward edges and back edges; there are no other retreating edges left out. If this happens then the flow graph is actually. It is

okay to include those cross edges, for example, here these cross edges can be included in the DAG because it does not violate the tree property. It is just that the retreating edges cannot be included and of course, all retreating edges must be back edges.

(Refer Slide Time: 19:49)

Reducibility

A flow graph *G* is reducible iff

- its edges can be partitioned into two disjoint groups, *forward* edges and *back* edges (back edge: heads dominate tails)
- forward edges form a DAG in which every node can be reached from the initial node of *G*
- In a reducible flow graph, all retreating edges in a DFS will be back edges
- In an irreducible flow graph, some retreating edges will NOT be back edges and hence the graph of "forwa[rd]" edges will be cyclic

So let us look at an irreducible flow graph; some retreating edges will not be back edges. We are going to look at some examples of this and hence the graph of forward edges will be cyclic.

(Refer Slide Time: 20:00)

Reducibility - Example 1

7 → 3, 10 → 7, 4 → 3, 10 → 3, and 11 → 1 are all back edges.

There are no other retreating edges in any depth-first search tree of this graph.

The rest of the edges form a DAG, in which each node is reachable from node 1.

Reducible graph.

Flow Graph

Let us take our standard example. In this graph, there are many back edges - 4 to 3, then 7 to 3, 10 to 7, 10 to 3 and 11 to 1. These are all back edges and it so happens that they are exactly the retreating edges as well. There are no other retreating edges which are not back edges; there are none at all. So, all the retreating edges are also back edges here. If you leave out the back edges then you can easily see that the rest of the flow graph is a DAG and every node is reachable from one. Therefore, from our definition this is a reducible graph.

(Refer Slide Time: 20:48)

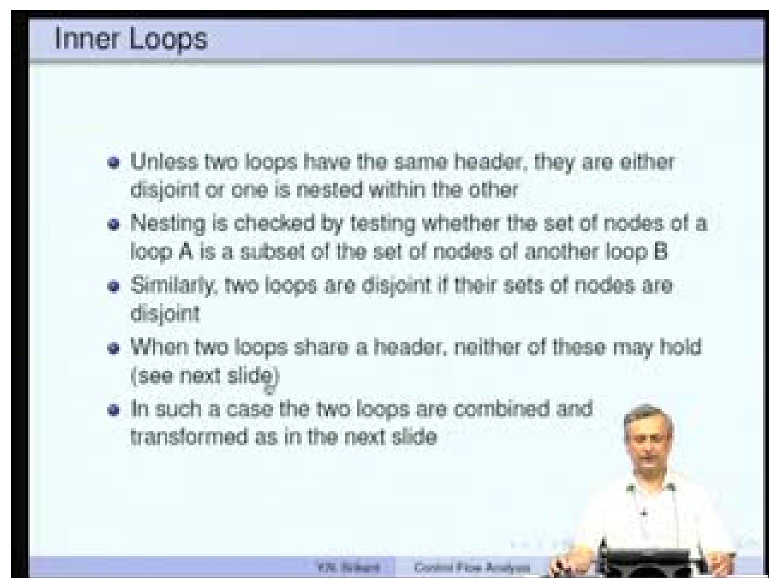
Let us take this example; this is an irreducible graph. If you do a depth first search on this, we start from 1, then go to 2 and then go to 3 and nothing more to do.

Now, 3 to 2 will be a retreating edge, whereas if we had started from one and we had gone to 3 and then to 2; 2 to 3 would be a retreating edge, but in this particular graph, 1 dominates itself ; neither 2 nor 3 dominate each other; there is nothing you can do. If I start from 1 then I can go to 2 and there is an alternate path as well - from 1 to 3 to 2. I really do not have to work through either 2 or 3 in order to reach the other node. We do not have any back edge in this particular flow graph. If you had a back edge then either 2 or 3 would be one of the tails, the other would be dominating the first one - either 2 would be dominating or 3 would be dominating 2; this does not happen. So, only 1 dominates 2 and 3 and neither 2 nor 3 dominate each other.

So, because of this there are no back edges, but the graph itself does not have only forward edges it is not a DAG anymore. If you include these 2 edges then the graph is cyclic and therefore, from our definition leaving out the back edges, the rest of the graph should have been acyclic, but it is not; therefore, this is not a reducible graph, it is irreducible. It so happens that if this graph is contained as a sub graph of any other control flow graph then that graph also become irreducible.

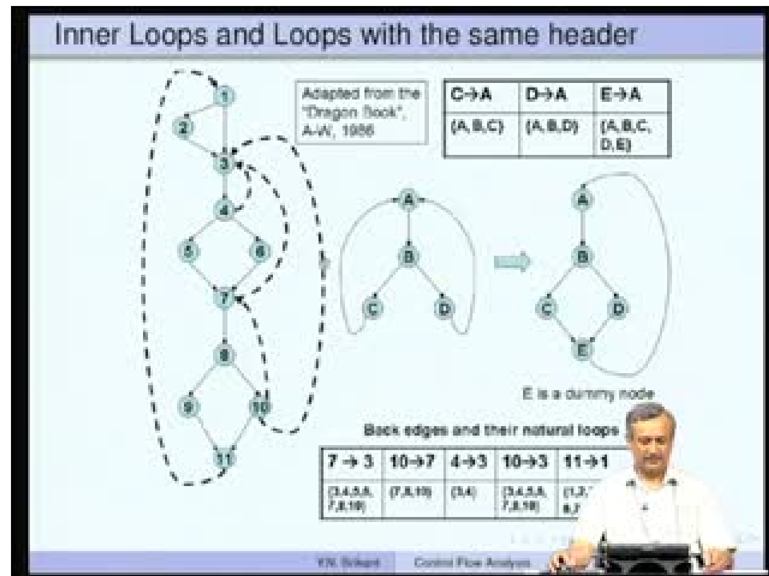
We are going to see a concept known as node spitting very soon. When we split, let us say node number 2 we get this graph. With this graph which is a reducible graph, let us say we do the depth first search. We go from a to b to c to d and then d to c is a back edge because c dominates d; all pass from a to d have to go through c. If you leave out the edge from d to c then the rest of the graph a, b, c and d will form a DAG; no problem at all. Therefore, this is a reducible graph.

(Refer Slide Time: 23:43)



The application of reducibility will be seen a little later. Let us go a little further and see how we can define nested loops. What are inner loops? They are nothing, but loops nested by other loops; unless 2 loops have the same header, they are either disjoint or one is nested inside another.

(Refer Slide Time: 24:12)



Let us look at an example. These are the back edges in their natural loops. For example, if you look at the back edge 10 to 7, 10 to 7 is here and you look at the back edge 7 to 3 - that is 7 to 3. You see that 7, 8, 10 is a strict sub set of 3, 4, 5, 6, 7, 8, 10. Of course, in this 3 is the header and in this 7 is the header. 7 is the header here 3 is the header here, but one is a subset of the other and therefore, this loop is nested inside this particular loop.

But the same is not true between these two. There are no nodes which are shared; these two are disjoint loops, but, 10 to 3 is again 3, 4, 5, 6, 7, 8, 10 so 7, 3 and 10, 3 are identical. There is no question of nesting here; they are identical loops. 11 to 3 nests all other loops because all others are strict sub sets of this particular loop set.

(Refer Slide Time: 25:26)

Inner Loops

- Unless two loops have the same header, they are either disjoint or one is nested within the other
- Nesting is checked by testing whether the set of nodes of a loop A is a subset of the set of nodes of another loop B
- Similarly, two loops are disjoint if their sets of nodes are disjoint
- When two loops share a header, neither of these may hold (see next slide)
- In such a case the two loops are combined and transformed as in the next slide

Y.N. Srikant   Control Flow Analysis

That is what we mean by nesting of 2 loops. Nesting is very easily checked by testing whether the set of nodes of a loop A is a subset of the set of nodes of another loop B. This is a very simple straight forward task. Similarly, two loops are disjoint, if their sets of nodes are disjoint. This also, I already had shown you how

Now there is something slightly different. When two loops share a header, neither of these may hold true and in such a case we may have to combine the loops and transform them as I show you now.

(Refer Slide Time: 26:09)

Inner Loops and Loops with the same header

Adapted from the "Dragon Book", A-W, 1986

C→A	D→A	E→A
{A,B,C}	{A,B,D}	{A,B,C,D,E}

E is a dummy node

Back edges and their natural loops

7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6,7,8,10}	{7,8,10}	{3,4}	{3,4,5,6,7,8,10}	{1,2, [illegible]

Y.N. Srikant   Control Flow Analysis

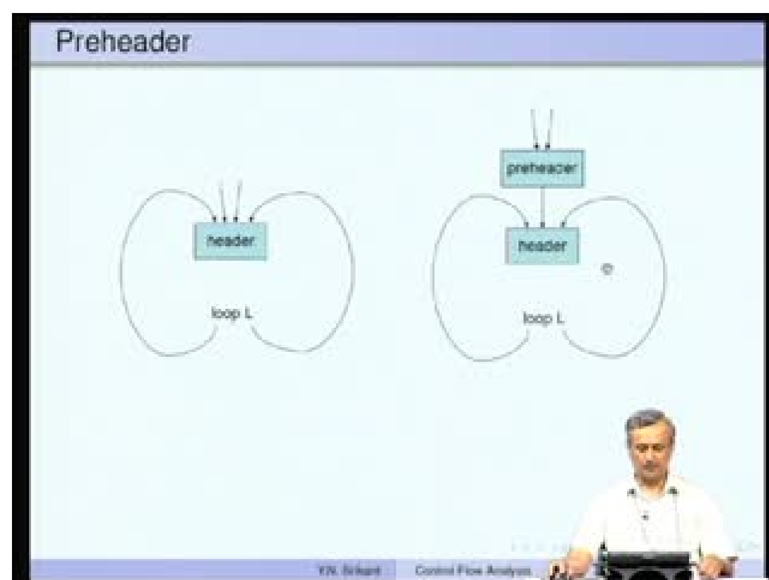
Let us take this structure. Before explaining this, let me hasten to add that when 2 loops share the header, one is nested inside another that is still possible. Take 7 to 3 and 4 to 3 and 4 to 3 is a subset of 7 to 3 and they share the same header; they are nested. 4 to 3 is nested inside 7 to 3 which is perfectly okay. What I say is it may not hold.

For example, in this flow graph A, B, C, D, there are 2 back edges C to A and D to A. This is one back edge and this is the other back edge because A dominates C and A dominates D. Now A, B, C is the loop here. How do I cover the loop? I start with A and I start with C; C's predecessor is B and that is all.

So, we have 1 loop here. Similarly, we have the other loop A, B, D here. A, B, C and A, B, D are 2 loops; they are neither disjoint nor subsets of each other, but they share a header. This structure is a very awkward structure. There were 2 loops sharing a header. We want to process them as a single loop. We introduce a dummy node, say E which has no statements - null statements and then we attach edges from C to D to this particular node E and the back edge from C to A and D to A is converted into a single back edge from E to A.

If you look at E to A, the loop will be A, E then C, D and B; all of them are in the loop. So, it is a single loop. This can be analyzed quite nicely. That is the reason, why we want to convert such loops into this type of loops.

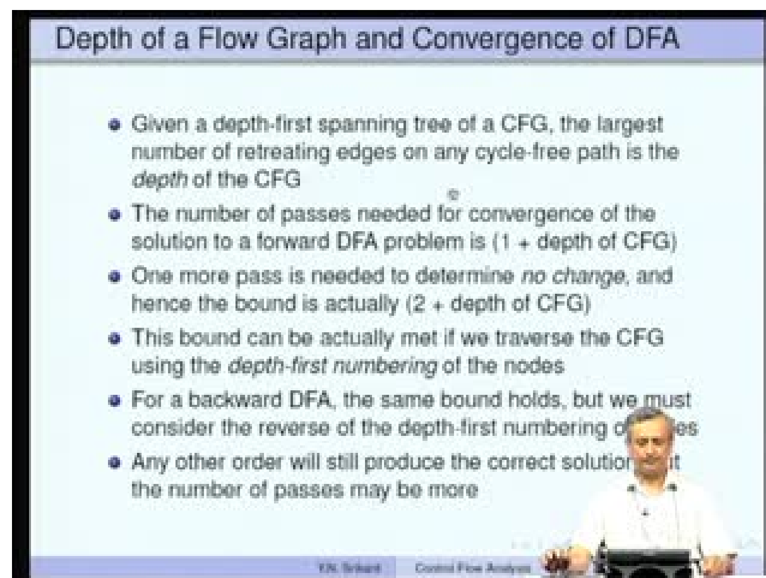
(Refer Slide Time: 28:10)



Then a concept of preheader must be explained. Suppose you have a loop with a header which is loop L and later when we do loop invariant computation elimination rather movement, we may have to move statement from the loop L to the outside of the loop. Where do we place those statements which have been moved?

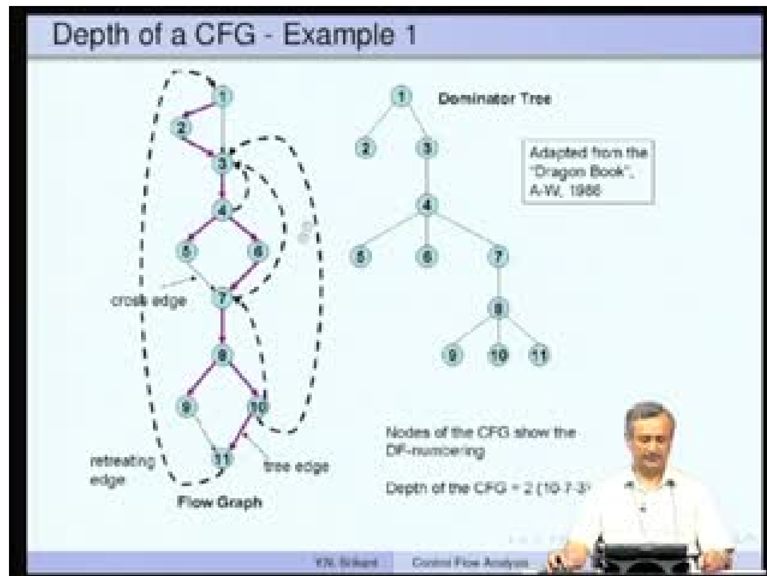
We create a new basic block called a preheader and place those statements there. There is an edge from the preheader to header, nothing else. The loop part remains the same and all incoming edges which are coming into this header will now go into the preheader. So, semantically this loop is exactly the same as this loop. It does exactly the same things but, the syntax structure is slightly different. It makes it slightly more convenient to process the optimizations.

(Refer Slide Time: 29:10)



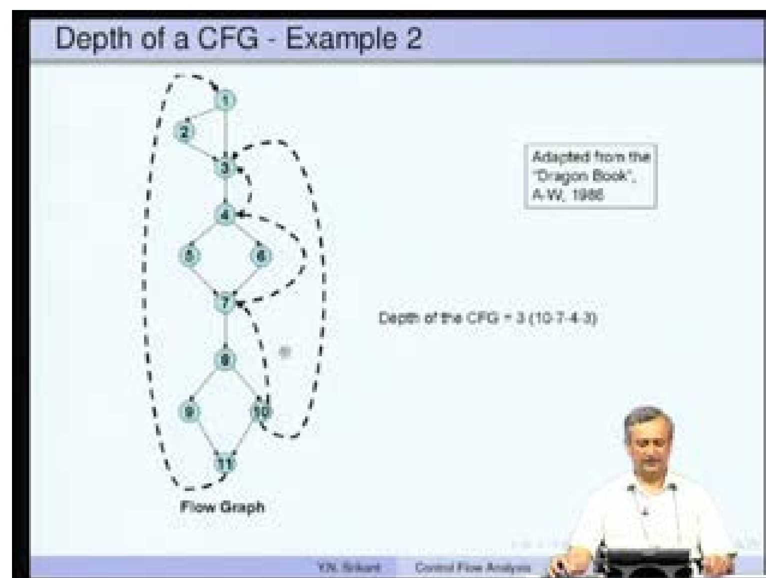
Now the depth of a flow graph is related to how many iterations we need for a data flow analysis problem to converge to the solution. If we are given the depth first spanning tree of a control flow graph, the largest number of retreating edges on any cycle-free path is a depth of the control flow graph.

(Refer Slide Time: 29:45)



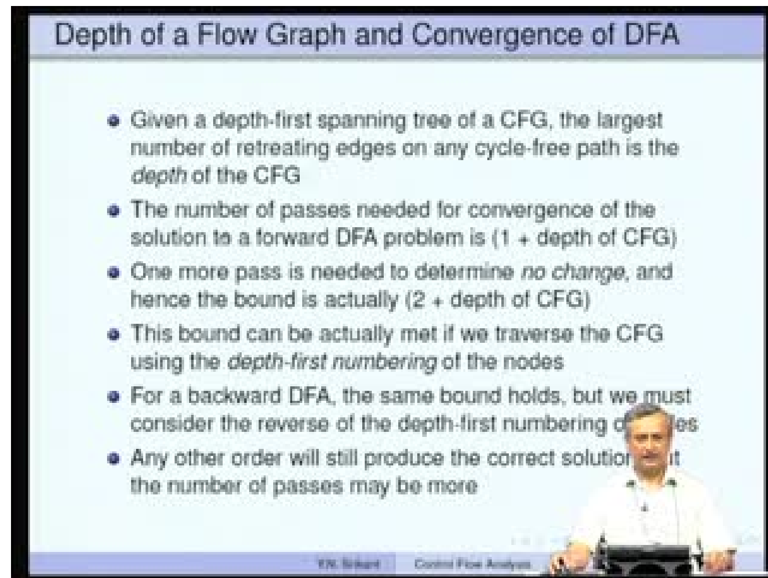
Let us see what this means. For example, we can just say we go from 7 to 3 or 4 to 3. Well then we have traversed just one back edge, but suppose we go from 10 to 3, we take 1 half to 7 and then another half to 3. It so happens that there is another straight half from 10 to 3, but this is the longest set of back edges that we can traverse when we go from 10 to 3. So, the depth of the control flow graph is really 2.

(Refer Slide Time: 30:24)



Now, in this example when we want to go from 10 to 3, we go from 10 to 7, then 7 to 4 and then 10 to 3. This makes the total as 3 back edges and therefore, the depth of this control flow graph is 3.

(Refer Slide Time: 30:42)



The number of passes needed for convergence of the solution to a forward data flow analysis problem is always 1 plus depth of the control flow graph and one more pass is normally needed to determine that there is no change and with this the bound is actually going to be 2 plus depth of the control flow graph.

In other words, if there is no deep nesting of loops then most control flow graphs can be processed very quickly. That is very nice. We do not have to actually perform too many iterations in order to get the solution to a data flow analysis problem. Now, the relationship with the depth first search tree: This bound can be actually met if we traverse the control flow graph using the depth first numbering of the nodes.

This is the use I was talking about. We have the depth of the control flow graph. We want to use as many iterations as the depth of the control flow graph plus 2 to carry out data flow analysis. How do we achieve this? Just take the control flow graph, do depth first numbering of the nodes and then use that particular order to traverse the nodes of the control flow graph when we do data flow analysis.

This ensures that we do not take more than 2 plus depth of CFG to converge to the solution. For a backward data flow analysis problem the same bound holds, but we consider the reverse of the depth first numbering of the nodes. That is very important. We must also be careful we are talking about reducible graphs. If the graph is irreducible we have to convert it to reducible graphs and then do depth first numbering and finally, traverse the CFG in that order, we get the 2 plus depth of CFG as the bound.

What happens if we use any other order to traverse the nodes of the control flow graph and perform data flow analysis? Nothing wrong, we still produce the correct solution, but the number passes needed may be more. We may not actually converging 2 plus depth of CFG number of iterations.

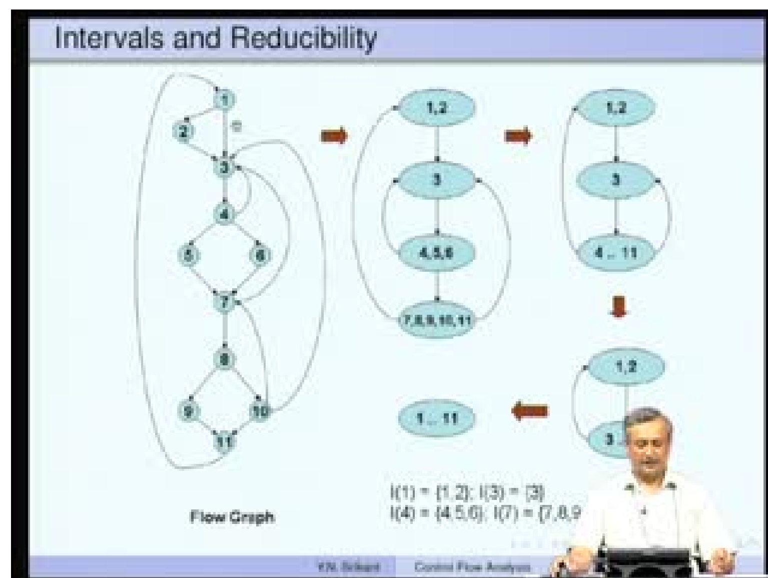
(Refer Slide Time: 33:34)

The next concept in control flow analysis is that of intervals. What exactly are intervals? What are they useful for? This is something we need to understand very thoroughly. Intervals have a header node that dominates all nodes in the interval. Again the best thing is to look at examples, but let us also look at the definitions and then go to the example.

Suppose you are given a flow graph G and there is an initial node n0 for the flow graph; there is also a node n of the flow graph. What is the interval with header n? This is denoted as I(n), n is the header of this interval. It is defined as follows.

The node n itself is in the interval; no problem at all with this. Then if all the predecessors of some node m not equal to n naught are in the interval then we add m into the interval; nothing else is in the interval. How do you construct the interval? This definition is translated into a small algorithm here. Start with I(n) equal to n that is number 1. While there exists a node m not equal to n naught, that is what we see here, all of whose predecessors are in the interval I(n) do: add m into the interval. So, we have just translated these three into a small algorithm here.

(Refer Slide Time: 35:24)

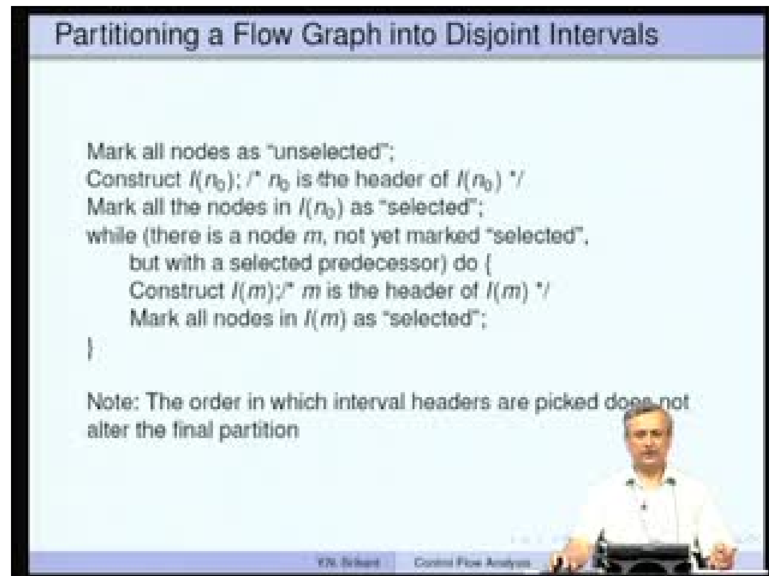


Let us see how this interval can be constructed. If you look at the node number 1, that is our initial node n0.

We add that into the interval. Then if you look at node number 3, it has many predecessors: 2 is one of them, 1 is another, then we have 10 and then we have 7 and then we have 4. We require that all the predecessors must be inside the interval already. We cannot put 3 to the same interval as 1.

Consider node 2; the only predecessors of 2 is 1 and 1 is already in the interval set. So, 2 can be added to the interval set. 1, 2 is an interval but, even then we cannot add 3 because 1 and 2 are in the set, but 4 then 7 and 10 are not in the set. We cannot progress beyond 1 and 2 for this particular interval.

(Refer Slide Time: 36:35)



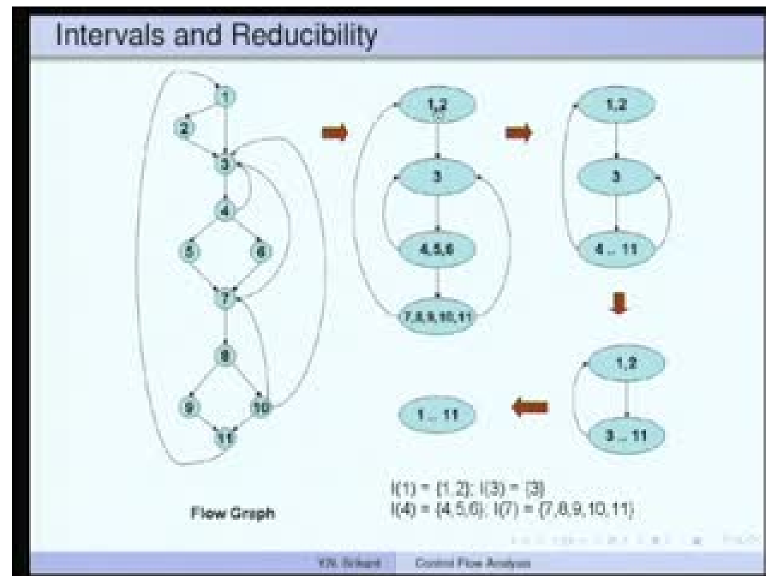
We know how to construct an interval, but we also need to learn how to partition the nodes of a flow graph into disjoint intervals. It is not enough to know how to construct one interval. We should learn how to construct many intervals and then once exhausted, the intervals must partition the nodes of the control flow graph into disjoint sets.

So start with marking all nodes as unselected and then construct the interval for the initial node n0. This n0 is always the header as well for that particular interval. Mark all the nodes in I(n); I(n0) is constructed using the procedure which we described a little while ago so and all those nodes inside I(n0) are marked as selected. Obviously 1 phase is over.

How to pick the next header? While there is a node m not yet marked selected, but with a selected predecessor; it is not necessary that all its predecessors are selected, if it is that way then we would have inserted into the previous interval itself.

Any one of the predecessors may be selected. That is sufficient. Use that as the next header and construct I(m); m is the header of this interval I(m). Mark all the nodes in I(m) as selected and we keep repeating this until all the nodes in the flow graph are visited. There is a note here which says the order in which interval headers are picked will not alter the final partition.

(Refer Slide Time: 38:37)



You can pick the nodes in any order to be headers, but the intervals will remain the same. We started with 1 and we built 1, 2 as an interval; then if you look at any other node which has either 1 or 2 as a predecessor because 1 and 2 are the only ones which have been selected so far; it is only 3, there are no others.

Of course there is 11; 11 has 1 as selected. You could also pick 11 and then go about constructing the interval for it, but let us pick 3. When you pick 3, you have a small problem you added it to an interval not problem but, the interval for 3 is a singleton set containing 3 you cannot add any other node into this particular interval because any other node also has many other predecessors.

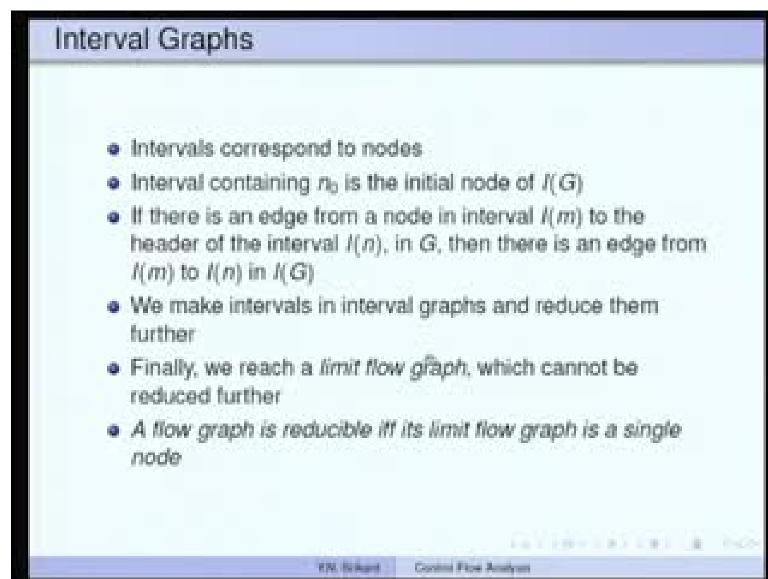
This is the trouble. What we do is, we just have 3 and then 4 is the next selected header. We have 4 and then 4, 5, 6 will be added into the interval and finally, we have 7 which has predecessors like this; 7, 8, 9, 10, 11 will all go into the same predecessor sorry same interval. In this manner we have this particular set of intervals {1, 2}, {3} then {4, 5, 6}, {7, 8, 9, 10, 11}.

Now, you can add really edges among these intervals. Whenever there is an edge from one of the nodes inside this interval to its header, then we add that edge here. This is the concept of an interval graph. We have an edge from 1, 2 to 3 because from 2 to 3 there is an edge.

And 3 to 4, 5, 6 because there is an edge from 3 to 4 as well and then 4, 5, 6 to 7, 8, 9, 10, 11 that is because we have 6 to 7 and then 4, 5, 6 to 3, 4 to 3 is an edge and then 7, 8, 9, 10, 11 to 3 that is because of this edge. This to this, because there is an edge from 11 to 1 and so on.

There are many of these. That is where it is now. We can construct intervals on this particular reduced flow graph of intervals. When we do that this and this combined together, we get a single interval 4 to 11 and these two remain the same. Then 3 and 4 to 11 combine together, to become a single node. Finally, 1 to 11 combine together, to become a single node. So, this is the interval structure.

(Refer Slide Time: 42:06)



Intervals correspond to nodes in an interval graph. Interval containing n_0 is the initial node of the interval graph and if there is an edge from a node in interval I(m) to the header of the interval I(n), in G, then there is an edge from I(m) to I(n). This is what I just now said. This is the way we construct the interval graph. We make intervals in interval graphs and reduce them further. Finally, we reach what is known as a limit flow graph which cannot be reduced further and a flow graph is said to be reducible, if and only if, its limit flow graph is a single node. This is an alternative definition of reducibility. (Refer Slide Time: 42:47) In this case, we reach this limit flow graph which is a single node and therefore, this entire flow graph here is reducible.

(Refer Slide Time: 43:03)

Node Splitting

- If we reach a limit flow graph that is other than a single node, we can proceed further only if we split one or more nodes
- If a node has k predecessors, we may replace n by k nodes, $n_1, n_2, \ldots, n_k$
- The ith predecessor of n becomes the predecessor of n_i only, while all successors of n become successors of the n_i's
- After splitting, we continue reduction and splitting again (if necessary), to obtain a single node as the limit flow graph
- The node to be split is picked up arbitrarily, say, the node with largest number of predecessors
- However, success is not guaranteed

This is about reducibility of a flow graph. What happens if a flow graph is not reducible?

(Refer Slide Time: 43:17)

Node Splitting Example

Irreducible graph; no back edge, only forward edges, but graph is cyclic; each node is an interval on its own

1, 2, 3, 2a, 2b, 1,2a, 3,2b, 1,2a,2b,3, 1,2b, 3,2a

For example, we saw that this flow graph is not reducible. It has forward edges; there are no back edges. So, this graph is cyclic and therefore, it is not reducible. We saw that already. If we have something like we cannot reduce this further, whatever you do even if you form intervals, you get one interval here, another interval here and the third interval here and we cannot reduce this further.

In such a case, we reach a limit flow graph that is other than a single node, we can proceed further only if we split one or more nodes. When we split a node there is no guarantee that the graph will become reducible, but there is a very good chance that it will become reducible.

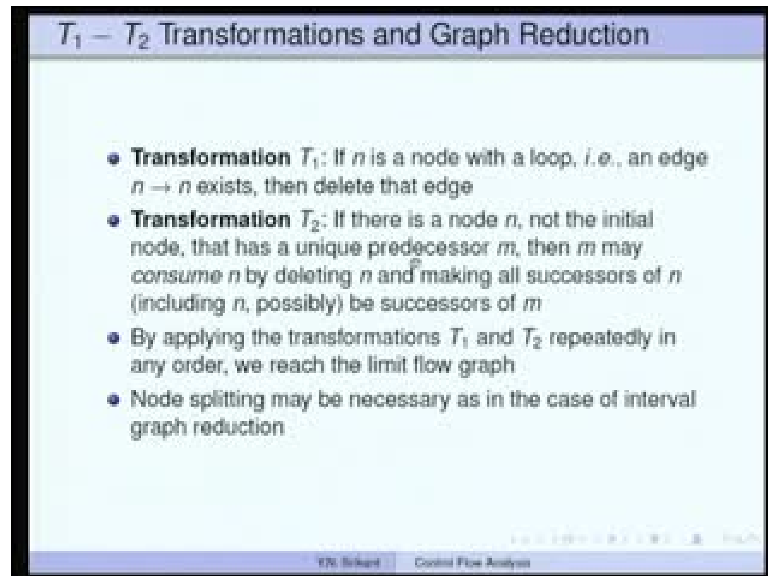
What is this process of splitting? If a node has k predecessors, we may replace n by k nodes n1 to nk. For example, here this node 2 has two predecessors 1 and 3. So, we create two clones 2a and 2b. If a node has k predecessors we may replace n by k nodes n1 to nk. The important point is ith predecessor of node n becomes the predecessors of ni only. Each one of these get exactly one incoming edge while the successors of n become successors of the each of the ni's.

For example, here this edge 1 to 2 goes into 2a and the edge from 3 to 2 goes to 2b. Whereas if we wanted the edge from 1 to 2 go to 2b, the edge from 3 to 2 would have gone to 2a. It is just symmetrical.

Then the successors of the node 2 - there was an edge from 2 to 3.There is an edge from 2a to 3 and also from 2b to 3 - 2a to 3 and 2b to 3. This is what I said here. After splitting we continue reduction and splitting again, if necessary, to obtain a single node as the limit flow graph. The node to be split is picked up arbitrarily, say we can choose any heuristics not that it helps a lot, but suppose we choose the node with largest number of predecessors then there is a very good chance that the graph becomes reducible.

Then this also increases the size of the graph; many nodes are introduced fresh into the graph. However, success is not guaranteed. In this case, we can reduce it. You construct the interval for 1. Then you include 1 in the interval, go to 2 its only predecessor is 1. So, 1, 2a becomes an interval. You cannot include 3 because it also has another predecessor 2b. Similarly, you start with 3 then 2b is included in the interval. There is an edge between these two because there is an edge from 1, 2, 3 as well. We can combine these two and make it a single interval. This is the limit flow graph. Similarly, this way also we would have reduced it to 1, 2b and 3, 2a and finally, 1, 2a, 2b to 3. So, limit flow graph would have reached here. So, after splitting, this irreducible graph has become reducible.

(Refer Slide Time: 47:05)



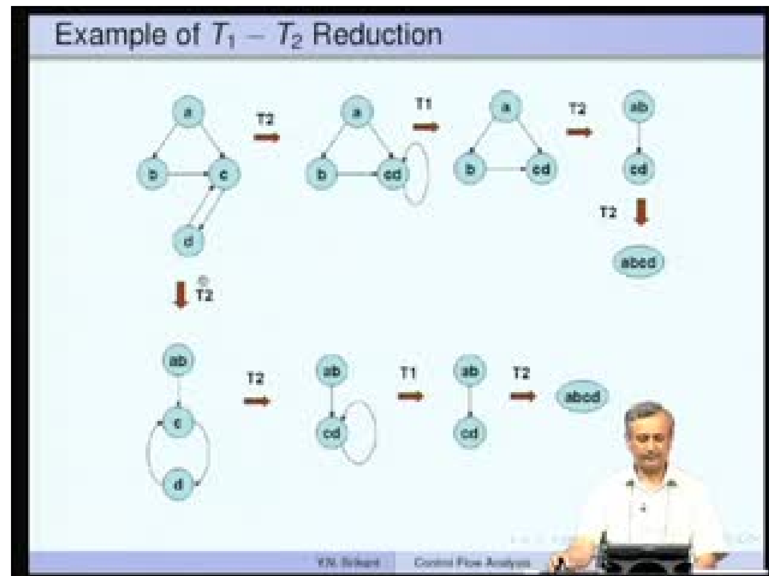
We start looking at one other type of transformation. We saw reducible flow graph definition using forward edges and back edges.

We saw reducible flow graph definition using intervals. Let us look at T1, T2 transformations and see how we can reduce graphs and define reducibility. The transformations are very simple. There are two transformations T1 and T2.

If n is a node with a loop that is, an edge n to n exists that is a self loop then delete the edge. The self loop is removed. This is the first transformation called T1. The transformation T2: if there is a node n, not the initial node that has a unique predecessor m, then m may consume n by deleting n and then making all successors of n (including n possibly) be successors of m.

By applying the transformations T1 and T2 repeatedly in any order, we reach the limit flow graph. If we reach the single node limit flow graph then the graph is said to be reducible. This is a third alternative definition of reducibility. Nodes splitting may be necessary as in the case of interval graph reduction.

(Refer Slide Time: 48:35)

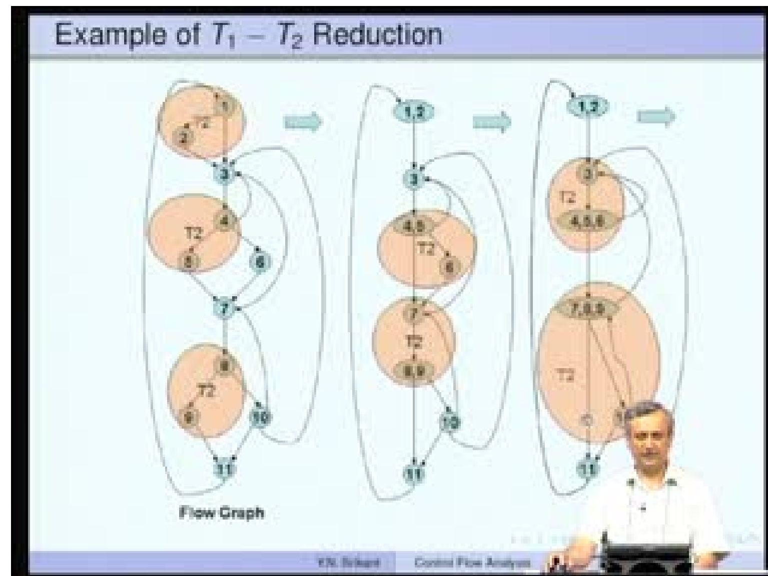


Let us look at an example. This graph is reducible. You apply T2 transformation and let us say combine here - this is one path and this is another path. We combine c and d because d's only predecessors is c. So, c and d combine into a single node and the edge form c to d now becomes the self loop on c. Now, there is a chance to apply T1 transformation. This self loop is removed and we get this graph.

Again apply the T2 transformation and combine a, b. You have combined a, b and then we already had cd. cd has this node; it does not have unique predecessors. You could not have applied the T2 transformation on cd directly, but ab can be combined. ab and cd can be combined because there is a unique edge predecessors for cd that is ab and you get the limit flow graph.

If you had taken the other path, you would have combined ab first then you have cd here. Combine cd by T2, then apply T1. Remove the self loop. You get the same abcd structure then you get same limit flow graph abcd.

(Refer Slide Time: 50:07)

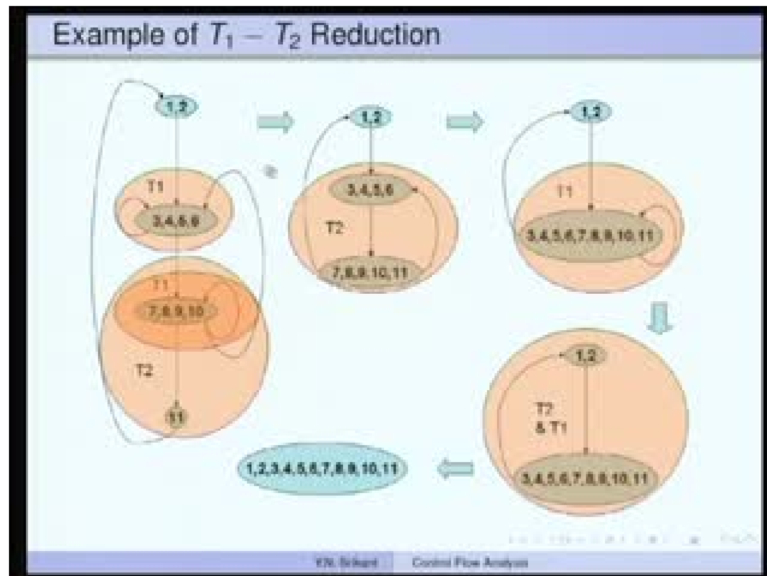


Let us look at a much bigger example to understand the T1 - T2 analysis. Here is our flow graph, the running example. You can apply T2 here because 1 is the unique predecessor of 2. You can apply T2 here also; 4 is the unique predecessor of 5. Similarly, 8 is the unique predecessors of 9. You apply T2 here also. Let us assume that we apply these three T2 transformations one at a time, but to save space I have shown them as a single step. You get a node 1,2 - combination of 1 and 2; 3 remains as it is; node 4,5 and then node 8,9.

Now the edge structure has modified slightly. Whatever was going out of 4 and 5, will now go out of 4, 5. There was an edge to 6; it is going out, an edge to 7; that is also going and then we had an edge from 4 to 3; that is also going. Similarly, here we had from 7 to 8; that goes 7 to 8, 9 and then 8 to 10, 8, 9 to 10, 9 to 11, 8, 9 to 11 etcetera.

Now, there is an opportunity to combine 4, 5 and 6 because 6 as a unique predecessor 4, 5. Similarly 8, 9 has a unique predecessor 7. These two can be combined to 7, 8, 9. These two nodes by T2 transformation can be combined to 4, 5, 6. There is further opportunity to apply T2 here. 3 and 4, 5, 6 can be combined. Another T2 transformation for 7, 8, 9 and 10 can be combined.

(Refer Slide Time: 52:05)



If we do that we get 3, 4, 5, 6 as one node 7, 8, 9, 10 as another; there is a self loop in each of these cases that enables T1 transformation. We applied T1 transformation in these places. The self loop goes way. Once it goes away, from here to here there is a single edge. So, the unique predecessor of 11 is 7, 8, 9, 10.

You can also apply the T2 transformation. If you do that you have 3, 4, 5, 6 and then 7, 8, 9, 10, 11. Now, you have an opportunity to apply T2. You get 3, 4, 5, 6, 7, 8, 9, 10, 11. This loop has become a self loop. Apply the T1 transformation and so you get this big node with 1, 2.

Again you have an opportunity for T2 first, followed by the self loop removal by T1. That reduces it to the big node 1 to 11. We have reached the limit flow graph. This is a single node and therefore, this entire big flow graph is reducible. So, T1 - T2 analysis is very useful in region formation.

(Refer Slide Time: 53:28)

Regions

- A set of nodes N that includes a header, which dominates all other nodes in the region
- All edges between nodes in N are in the region, except (possibly) for some of those that enter the header
- All intervals are regions but there are regions that are not intervals
  - A region may omit some nodes that an interval would include or they may omit some edges back to the header
  - For example, $I(7) = \{7, 8, 9, 10, 11\}$, but $\{8, 9, 10\}$ could be a region
- A region may have multiple exits
- As we reduce a flow graph G by T_1 and T_2 transformations, at all times, the following conditions are true
  1. A node represents a region of G
  2. An edge from a to b in a reduced graph represents a set of edges
  3. Each node and edge of G is represented by exactly one node or edge of the current graph

Let us see what regions are. A set of nodes N that includes a header, which dominates all other nodes in the region; that is a very simple definition of a region. All edges between nodes in N are in the region, except (possibly) for some of those that enter the header. I will have to give you an example for this.

(Refer Slide Time: 53:58)

Region Example

This arc corresponds to 2 arcs, CA and DA. Hence, the predecessors of T, the header of S in V are C and D

For example, here this is the region structure. I will tell you how. We get this by the T1 - T2 analysis. For example, take this node R which consists of the 2 nodes C and D. The edge from D to C is not inside the region R. It is actually in the next region S. This is

what I meant. There you can leave out some edges from the nodes to the header because they fall outside that particular region.

So, all edges between the nodes in N are in the region except for some that must enter the header. You really cannot leave out edge from C to D. You have to leave out only the edge from one of the nodes to the header when we combine these nodes.

All intervals are regions, but there are regions that are not necessarily intervals. This is something very important. So, a region may omit some nodes that an interval would include or they may omit some edges back to the header. (Refer Slide Time: 55:20) For example, in the previous case 7, 8, 9, 10, 11, the big one here, this entire thing became one interval as we had seen, but 8, 9, 10 could be a simple region on its own. Just this small one could be a region; 8 dominates 9 and 10.

So we include these 2 edges in the region and that becomes a region. So, region structure is more loosely defined than an interval. Region may have multiple exits. Again, that is possible. As we reduce a flow graph by T1 - T2 transformations, we find regions. A node represents a region always in the reduced graph. An edge from a to b in a reduced graph represents a set of edges. Each node and edge of G is represented by exactly 1 node or edge of the current graph.

That is the example I was trying to show you. We did this T1 - T2 analysis on this. We have the region AB obtained by the region T obtained by combining A, B. Regions R obtained by CD and then self loop removal gives us region S and then further T2 transformation gives us region U and then self loop removal again in this gives us region V. This is the region structure we have uncovered. When we did T1 - T2 analysis and if you look at the edges between T and S; this is T and this is S. This particular R actually corresponds to 2 arcs.

So the arc from S to T corresponds to the arc from C to A and also from D to A. These are the two arcs that it corresponds to. This is an example of a region. This is the end of control flow analysis. In the next lecture, in the next class, we will consider how to do possibly optimizations and then data flow analysis on regions.

Thank you