웹 소프트웨어 신뢰성

✓ **Instructor: Gregg Rothermel**
✓ **Institution:** 한국과학기술원
✓ **Dictated: OES** 봉사단

Okay, well, where are we?

Last time we looked at dominator information that is used in certain analysis and one that it' used in is for subsequent analysis to calculate control dependencies as you all have seen from the reading for today.

So now we are going to talk about the control dependence algorithm.

I asked you to read the section of this representation paper pertaining to that and the first few pages of this for Auntie Augustie and Warn paper, which is, which gives the real algorithm and algorithm in this representation paper is the algorithm that's in here but I think is expressed in a way people could actually understand maybe.

I remember we had real trouble understanding this algorithm.

Because they did nothing in here called algorithm, algorithm one that is described.

Okay, their motivation, they were into, optimization, which is as I said where many of these came from.

And I think this paper predates work on soft engineering that has used.

There has been some work on soft engineering things using the PDG, and we will look at some of that.

But there is this optimization, and they were also particularly interested in paralization of programs and it turns out that their PDG representation could help with that.

They say some of that in the paper.

So now let's finish data flow for next time, you can try and do control dependence for the pile of control dependence algorithm.

That will require that you first calculate post-dominator information, so you'll have to do both of those to get those problems.

So, actually I have already talked a little about the motivation.

And now we are going to talk about the algorithm.

Now, informally, and supporting that word of informalism important, here.

We will say a statement S1 is control dependent on a predicate statement S2 if the outcome of S2 determines whether S1 is reached in the control flow.

That, you will see that statement has a little bit of a problem in it, a little bit of ambiguity.

So we will need a more formal definition, but informally.

S1, is control dependent on this predicate statement, if the outcome of this predicate statement determines whether you reach this.

And certainly if this is false, you reach it, and if it's true, you don't. Okay?

Likewise, this predicate determines whether you reach this.

And determines whether you reach this.

And this predicate determines whether you reach this.

As it turns out, the imprecision in this definition is that it might also lead you to conclude that this predicate determines whether you reach this.

And in a sense, it plays apart, because if you take this path, you definitely, won't.

If you take this path, you are on the way to reaching it.

But there is something more immediate, that determines that.

So we'll more precise definition will have us stated this is control dependent on this, and this is control-dependent on this.

This on this, and this on these two.

So we will look at that more precise definition now, and see how that works.

Also, the, uh, you will see later, you've seen in reading that we also pretty much conclude that other things are control dependent on entering the program.

So we could say that these others are control dependent on entering, and this things end up being control-dependent on that.

B6 and on B1.

So how we calculate that.

Oh, well actually, before we calculate it, here's the more precise definition that I think is a verbatim from the FOW paper we call it.

So this time we use X and Y our nodes, in the control flow.

Control flow graph. We will say that Y, and Y is here.

It's going to be downstream and flow.

We will say that Y is control dependent on X, if there exists a directed path P, call it a directed path P, from X to Y, with anything on that path, post-dominated by Y,

And X is not post-dominated by Y.

So if you think about that, anything on here is post-dominated by Y.

It kind of means, okay, if X is not post-dominated by Y, remember the definition of post-dominator.

I'm going to go back to effects if Y post-dominates X, then every path from X to exit, must go through Y.

If Y does not post-dominate X, there must be some other path to exit that does not go through Y.

Okay, so that's where we're saying there exist one path from X to Y.

So, there's going to be two paths. There get to be two paths in order for X to not be post-dominated by Y.

There has been another path to the exit.

And that's that.

That gives us item 2 in the definition.

But Item 1 says there has been a path from X to Y, not just any path, but it has to be such that any Z on this path is also post-dominated by Y.

If Z were not, there was either or not, there's some other edge to exit from Z, then Z would not be post-dominated by Y.

And in fact, Y cannot be control-dependent on X, it might instead be control-dependent on Z.

So it's truly gets the idea that immediate control dependence accessor here.

Now, another informal way that it's kind of hard to wrap your head around that definition you have to think hard.

And you have to say, well let's see, this is X, this is Y, Z's, we are going to do that on this in a minute.

But there is also an informal way to think of it that works better than the previous Y.

Y is control dependent on X, if there are two edges out of X, such that taking one edge all gets to Y, and the other may not get to Y.

But you have to have take one edge always gets to is Y.

Let's first try that informal definition on this. Okay?

Is this control-dependent on this using the informal definition?

B4 is control dependent on B1, if there are two edges out of B1 that's two.

And we're traversing one edge always leads to is B4, it does.

And the other may not. In fact it does not. All it requires to say is it may not.

So this is control-dependent on this by that informal definition, okay? B5, is B5 control-dependent on B1?

B5 is control dependent on B1 if there are two edges out of B1.

B1, one edge always gets to is B5, and the other may not. It may or it may not.

So that's why we can say this control dependency exists, okay?

The B2 case is going to be similar to B4 case and there is two edges, one gets there always one may not in fact does not.

And here's the interesting one.

B1, B3. The definition says B3, control dependent on B1, if there are two other edges to B1, there are such that one edge always get to B3, there isn't any edge that always gets to B3.

Because you can take this route.

So it's getting at the notion of, in some sense, you have to go true to get to this.

But it's not immediately determining whether you get to this.

The immediate determinant of that is this. Okay?

We are in immediate to determine to that.

So you can use that informal one to try and get things, but more formally a thing that we can use in algorithm is this statement.

So now let's look at that one for all of these edges.

Okay, so again, X and Y, let's first try B1 equals X, B4 equals Y, okay? We are going to say Y is if.

Item 1. There's a directed path from X to Y, there is with any nodes in the path, excluding X and Y post dominated by Y.

Well, the seven nodes in the path are null-set.

So it's trivially true that any node in the path is post-dominated.

Makes sense?

Trivially true, there are no nodes there. S

o any node in there is post dominated.

And last, X is not post-dominated by Y, now we haven't drawn the post dominator tree, but I mean you can see that there's another path to the exit that doesn't go through Y.

Soy can see that's true.

So, Y is control-dependent on X, or B1 I meant to say B4 is control dependent I'm using the control dependence on B1, so we are looking at the interesting ones here.

Alright? Now, let's do B1 and B5. Okay?

Start this off, now what do we think first?

Is there a directed path from B1 to B5 in the graph? Yes, there is.

It goes to B4. Are there any nodes Z on that path? What's the node? That's on that path? B4.

Is B4 post-dominated by B5?

It is, because you can't get to the exit from B4 without going through B5.

So the case one halts.

Part one of the definition.

Second, is B1 post dominated by B5? It's not because there is another path to exit from B1.

So that is control dependence also.

Let's do B1 to B2. Someone else go through that definition step by step.

Start with Pablo.

The way I was doing it, okay? Just step by step.

And I'm asking the question is B2 control dependent on B1. Okay?

(Student answering) That's right, yeah.

Very good, that's it.

So B2 is control dependent on B1. Okay?

And let's do B3. John, oh what I mean by B3 is B1 B3.

No let's do B3 and B2 first. Okay?

Let's do B2, B3 first. B2 is a predicate node, so we can ask is B3 control dependent on B2. (Student answering) Okay.

Post-dominated.

Yes, B3 is control dependent on B2.

What now should we do?

Let's try more fun one for a minute.

I want to know if B1 is, if B3 is control dependent on B1.

Joo Cheol, Student speaking.

There's any no B2 is have another path to B5 so one is unsatisfied.

That's right. Put in these terms B3, B1 is not postdominated by B3.

Right? Good.

So that one doesn't have.

That's what we saw when we use this one two edges out traversing one there's not on edge that always leads there. Ok? What's another edge we can look at.

Well we can certainly look at B1, we can certainly ask it's not an edge it's a path, it's a B6 control dependent on B1.

Student answering, there is direct path from B1 to B6 so on the path there is two nodes 4 and B5, both of them post dominate B6 so constraint one is gone and second x is not post dominated by Y, this is also truth so B6 is control dependent on B1. Good.

What else can I do is a, Well I guess I will ask is exit control dependent on B2.

Student answering between B2 and exit, there exists directed path and there is no Y that can use as the Y is exit and Y itself can be, yeah Ok B2 to the exit, oh actually there are other two directed paths aren't there?

But you'd have to consider the two paths in turn.

But let's start with this I think let's start with this one.

With this path ok? So Y itself is exit so can you say B2 is predominated, post dominated by exit is it ok, yes B3 is post dominated by exit.

Then that means the first one halts, that does, there exist path that one will halts then what about this?

B2 is also post dominated by Y, yes, into the exit, so the second one does not halt.

That's right. So X is not controlled.

That's right. That's true. OK. Good.

What else can we do.

Well, I think I can do, let's do B6 and entry.

You're the only one left. Student answering

There is no intermediate for that, there's no what? I'm sorry. Intermediate.

It is directly entry oh sorry, I mean entry and B6, there is direct path, B1 and B4 B5 are not post dominated by Y so first condition is not satisfied.

That's right. So it's not contrarily dependent. That's right. Good. What is x dependent, what are you know the ones like, B1 kind of dangling there.

And we didn't put entry in exit in, and really there are I can summarize them as E and X, really what we will say they're dependent on this entering the program., ok so if you execute the program and that's where in front of ()21.30 we'll see this in a

minute, they will introduce one more node S and they will connect S by two edged entry and a false edged exit.

And all these ones that are kind of top level dependencies get hooked up to S.

Ok and the only other thing we can add to this are the conditions that caused the dependency are from B1 to B6 it's B1 false.

From B1 to B2, is B1 true, B2 and B3, it's B2 true, oh there's one other thing we didn't get here do we.

We didn't get the B2 false dependency with B5

B2 to B5, there is a directed path, there's no nodes in it so one is trivially true and B5 is not post dominated by B2 so that produces an edge over there.

That's the false edge.

B1 to B4 false, B1 to B5, false. And then kind of trivially(?) 22.25 we say start true there ok. Yes?

There is no intermediate nodes between two nodes such as the case in B1 and B2, there is intermediate node, Should we regard the first condition as true? Yes, what we say then when we read this, there is existed directed path, such that if there are any Z's in the path, you can read it like that ok? They are post dominated. So if there is none, it's trivially true. Ok?

There's some other control dependent too.

Did I miss another one? Yeah, B2 and B6. Yeah, very good.

Thank you. B2, B6 again. Let's do that.

There is a directed path, here is the Z, it's post dominated by B6, and there is and B6 is not post dominated, does not post dominate B2 because there is a another way around.

Thank you. B2, B6 Ok. See if that matches what I have.

Oh this is ok, this is the FOW method.

Oh now what we are going to do with FOW aren't we. OK.

That was going through it.

Using your brain and the definitions.

Obviously we'd rather have an algorithm and That's what Fromiti.N.Warren give us so they say augmented with the start node as done there.

Construct the post dominator tree.

Let's see. Hang on a second. Oh good. I can. I don't have to put the PDT up.

So that's the postdominator tree and now there is the steps in the algorithm and those steps are on page misfigured 12 , page 12 is the CG algorithm.

And there's these four steps, in part 3 3a, 3b, 3c, and d 3d, so find S a set of edges such as B is not an ancestor of A in the PDT.

What does imply is iterating through to the edges in the CFG.

So we are going to consider every edge and ask the question whether B is an ancestor of A or not an ancestor of A.

So let's see. Let's start from the start node.

We are using the CFG now. Keep that in mind.

So one of my edges start to E.

So start equals A.

You've get to replace these things in here. A is start.. B is E is B, an ancestor of A. E, an ancestor of start in the tree is B an ancestor of A in the postdominator tree now ancestor means poor right?

So E an ancestor of start in the postdominator tree.

It's not. OK? So that's going to be actually will leave A and B here.

So we are finding these edges it's a set of edges.

So here is the set we are building, and we've just put in start to E. OK?

Let's work on the other edge from start. Start to x, is X an ancestor of start in the tree?

Yes, it is.

And we are in just in the ones that are not ancestors.

So we don't take that. OK? Now let's go to E and 1. Is 1 an ancestor of E in the tree? It is.

So we don't include it.

1 and 2. Is 2 an ancestor of 1 in the tree? NO, so that's where we're interested in.

1 and 4. 4 an ancestor of 1 in the tree? It's not.

2 and 3. Is 3 an ancestor 2 in the tree? No. 2 and 5. Is 5 an ancestor 2 in the tree?

No. What did I, did. I didn't do 4 and 5 yet did I. 4 and 5. Is 5 an ancestor 4?

Yes, so we don't care about it.

Is 5 to 6. Is 6 an ancestor of 5. In the PDT.

Is 6 an ancestor of 5? 4 and 5? Did I miss that? Here 4 and 5, I mean have to ask. consider edge 4 5, and ask the question, is 5 an ancestor of 4 in the PDT and the answer is it is,

And we are confronted with ones that aren't.

I know there is negative in here, it makes it hard, but so that's why we didn't include it.

And 5 and 6, 6 is an ancestor of 5 and so we don't want it.

And the last, rather several edges to X, 6 to X, Xs in an ancestor 6, 3 to X, X is an ancestor of 3.

Start to X, We already did it. OK? So that was step A.

Find set of edges X.

Such that the synch of the edge is not an ancestor of the source in the PDT.

No I re find those ones. 1,2, 1,4, 2,3, 2,5 they are all immediate predict outcomes right?

If the thing is an ancestor well there are going to be things, things that are controlled dependent on, that's what I'm going to say. Ok? Step 2.

StartE 12, 14, 23. Step 2.

Now we go to the table.

And I think I can raise this in right in that.

I make this table of my edges that are in set S.

Now we are going to iterate step B iterate through these edges for each A, B in S, find L, the least common ancestor, the closest common ancestors what that means right of that.

So in the PPT, start an E. What's the common entry? Everyone see that? This isn't

common, this isn't common.

The one that's common is X, is when you go up and meet, okay?

So this common is X.

1 and 2? X. 1 and 4? X. 2 and 3? X. 2 and 5? That doesn't always happen.

We'll do an example when it doesn't but this is a relatively simple one and that was L okay? Step C.

We're going to literate from this paper again.

Consider each edge and its corresponding L..

Now the edge, they say, each edge A,B. okay?

That's what it says in the definition. Consider each A,B as the corresponding L reverse backwards in the PPT from B to L.

Marking each note visited but mark L only if L equals A.

You've get to get your head around that but let's just do it.

Here's the edge start to E. We're going to consider each note, we're going to traverse back from E to X, marking each note visited.

So when we do that, um, I'm trying to remember from mark E.

I'm pretty sure we do.

Let's just try that and see.

And we start from E. and the next one going backwards is 1 and the next one is X, but X does not equal A so we don't mark it.

One and two traverse backwards from two to X.

Which ones do we mark? Just two there's nothing else there.

This does not equal X. As a matter of fact, these first ones never equal X.

So we're never going to include L in the set and in these okay? One to four or, traverse backwards from four to X.

What do we get? Four five and six.

Traverse backwards from three to x. three traverse backwards from five to x.? five and six. Okay? That was step C. Statements representing all marked notes.

I'm going to go through this again.

Consider the marked notes.

The statements in there are control dependent on A, having the condition that is on the edge, A, B. So E and one are control dependent on Start and the condition that's on the edge from start to E is true.

I know that's not up here right, but basically, E and one are CD on Start True.

E and one are both… I can write start… on start being true. Okay.

Two is a marked note here. Two is control dependent on one being.., I've got to look back,,,

I thinks it's one true. We'll check that in a minute.

Four, five and six are dependent upon one being false, I think it is.

Three is dependent on two being, I don't know, I think it's true.

We'd have to look back again at the CFG.

And then five and six are dependent on two being false, I think.

Might have those trues and falses wrong so write in pencil.

And then you can build the CFG and the CDG on that. Let's see.

Start, true, one, true, false, one true, one false, two true, two false.

Okay I guess I got that right. And then, what comes next, create the graph.

So, control dependent on start true.

Wait a minute, I won't put the true up there, E and one are control dependent on start being true.

Two is dependent on one being true.

Four five and six are dependent on one being false.

I should be putting these edges labeled in. three on two being true. Five and six on two being false.

So, I drew it differently than what is on the graph, but that's what we had before, once Ji-min corrected me.

That's the algorithm,, that's the graph.

There're plenty of tools to implement this algorithm now.

And we'll end up seeing what's all of these are used for in a class or two.

Now, I don't include the algorithm in here for regions, I mentioned a little bit what regions are.

I guess I do show the graphs with regions.

I'm just showing you this so you know that they exist but I don't think any of our uses are going to make any use of regions, so you don't need to know the algorithm for putting regions in.

But what Freddy Evinston Warren wanted us to do with regions was find a way to group nodes that had the same sets as the controlled dependencies.

And so, I don't' think this is… I'm going to go right to the end.

But there are algorithms that does this in some steps.

And the end result is regions in capsulate sets of control dependencies.

So region six, well no let's start with an earlier one.

Region r 1 represents everything that is control dependent on start true.

Region r 2 everything that's on 1 true.

Region r 3 anything that's on 1 false. R6 represents things that are on 2 false and 1 false.. Then there are some.

And what they ended up doing, I believe, is finding some optimization algorithms that visited some regions in the graph and did some operations at those regions.

That's all you need to hear about regions.

We'll end up working with the graphs without them, like that. Okay?

Now, a program dependence graph combines control and dependent dependencies.

And that's where they end up getting.

The whole data dependence notion had been explored earlier and there were algorithms for that.

We've looked at those.

But you can take two programs and add data dependence edges representing computations that are in here and you'll have a program dependence graph.

So I don't know what data was in here.

This was a boring program. Where is it.

There were a couple data dependency programs in here that we could add to this graph.

For instance, b2 to b3. B2 to b5.

Actually I don't have to make it down, it's a different color.

B2 doesn't reach b6. B5 to b6.

There's two of those.

No there's one of those.

That's about it I guess.

There weren't that many data dependencies in that stupid little program.

But that's a program dependence graph.

Combining those two. There's also a variance of these that also draw in the control flow between notes.

So it all depends on what you're using it for.

There's variance of these at different dependency too.

You'll see one. These are definition to use or flow dependencies.

There are some versions of these that use definition to definition dependencies.

For certain optimization a def. before def.., if you got threaded programs, match. Okay.

All right we'll do one more.

Now first off, we're thinking about the control dependencies.

The contents of the note don't matter.

This is three address code, we've got labeled branches indication there are some flow after that.

Ignore the fact that there's no go to, that's been abstracted out along the edges, or if a statement, it's been distracted out.

We don't see the code for that.

But we can build a PDG for this I think. The first step, I will do control dependencies first, and the first step is to make the post dominator tree.

Now, going through that algorithm, which will take some time, let's try and make a tree just by thinking about it.

I feel I remembered to augment my graph, and let's put in an exit note, an explicit exit note, and I won't bother an entry note.

We'll let this one be the entry note.

But I did want to put in an exit.

Hang on a minute.

My answer did put in an entry note, so I'm going to do it.

So my answer will be the same.

The graph I gave you there didn't have an entry and an exit so I've added them and now we put in a S note with its true and false labels to entry and exit.

Now we're ready to begin with the post dominator tree.

Oh boy. Well, a post dominator tree shows, I'll get it started, it shows which immediately post dominates things.

And at the very root of it is the exit note, always.

And so the things we're going to see on the x are things x immediately post dominates.

There's no other post dominator that slips in between.

So what does that work for you? What are some immediate things? What are some things that x immediately post dominates? B6 okay? If you think about it, from b6 to x, there's no other path. And x is x. b6 to x is under there.

What other things does it immediately post dominates? Start definitely there was that.

Turns out there's one more here.

Even though immediately it doesn't necessarily have to mean that there's an edge attached to it, it means that nothing between x and that… It means the immediate set is the set A such that there's a path from a to x and nothing else on that path post

dominates.

So let me do this one for you.

Here's b5. And b6, what am I saying here. I have b5. I don't have b6 post dominated. Hmm. Did I do that wrong?

B6 does dominate b5.

So I did something wrong in my picture here.

This is screwing me up.

But basically every path to exit, even though there's a loop, every path, you always come back to b5.

So every path, that's going to be b6. I don't know that's probably going to mess up my whole answer but, let's see what comes under b5?

We can start asking about these notes.

Does b5 post dominates b2? It does.

But is there another post dominator in between? B3 yeah.

And the immediate one above that is b3.

But, is there another post-dominater in between? B3. Yeah. Above that is B3.

So, we can run the argorithm, and we get this more argorithmically, obviously. B3, B4, well.

Does B3 post-dominate B4? Any path from B4 of the X has to do with B3? Does B3 post-dominate B2? Any path from.. Remember.

This edge is going here. So, any path from B2 to the exit must go to B3? B2, B1, B1, entry.

That's the post-dominating tree. Now, we are moving to the algorithm.

S: set-up edges of A, B, since B is not an ancestral B in the PDT.

All right. Let's go through the edges one by one.

Start to E, S to E.

If you want, read A again. (A student reads the part) Say again, loud? That's right.

Therefore, we shouldn't do S to E. If as E is not ancestral B. E to B1?

We shouldn't include one, because E is an ancestral B. E to be one? We should not include, because B is an ancestral B.

B1to B2? Not include B2 to B3? Not include. B3 to B4? Not include. B4 to B3? Oh, yeah.

I think I got the backwards.

You're right. Hmm, wait a minute.

Let's think this through carefully. B3 to B4 find a set such that.. Oh, such a B4 is not an ancestor of B3.

Right. These trees are ancestoral Before, but it is true that B4 is not an ancestral B3.

So, B3 and B4 does go with.

But, what about B4, B3? There, you have do ask, "Is B3 an ancestral B4?" and it is, so that one doesn't go in.

B3, B5? Okay. B5,B6? B6, exit? And, what did we miss? We missed S, B6 and we missed X exit.

But, that one doesn't get in, either.

Did I miss one? 'Cause it feels slim. B5. What was that? Oh, thank you, of course. B5 to B2. B5 to B2. B5 to B2, and, is B2, an ancestral B5? No. So, the one what I had in here, which was incorrect and it was B5 and B6. Yeah, that's our set.

That's just step A.

Now, for each one in there, we're going to find L.

The least-common ancestor. Okay? Least-common of S and E? X. Least common of B3 and B4? B3. B5 and B2? B5. So we've got, remember.

There's going to this case where we L equals A, we do something.

Here's the case where that happens.

L equals A. I will start with that. L equals A in that case.

The other L does.. wait! Uf! L equals A.

Two cases. Now, consider each edge in these sectors, three of them, and do the marking.

So, we're going to go backwards.

And we're backwards from B to L.

So, backwards from E to X marking things. What do we mark? (writes something on the board) Okay? And X. Who marks X? Nope. Okay. B3, B4. We can go backwards from what to what?

Yeah, you're going to think about that. B4 to B3, what do we going to mark? B4 and B3. Okay? B5,B2? Or in backwards, from what to what? B2 to B5. What gets marked? Two, three, and five.

All right. And the control dependencies, statements representing these nodes are control dependent on S, having the value true.

Statements in here are control dependent on B3. B4, having the value true.

Statements here are control-dependent on B5, false. Okay? And doing that, we can come up with the graph.

It's going to start with S. S has a lot of children.

All label to, I'm just going to leave them true.

They're all true. And then, we've got B3,2 goes to B4, b4 is already in there. No, it's not. I did something wrong, I think.

Ture does not have full. B1, 2, 3, 5, and this..B3, 2 and then on B5 false, and B3, B4 and B5.

Let's think about a little of this. Just a little bit.

B5 is control dependent on its self-being false. Self being-false.

B5 could be control pendant of itself, if there's a path from B5 back to itself.

That always reaches to B5. And one that does not, obviously there's one that does not, or may not, does not.

And there's a path that always reaches it. Okay?

Then, that's why it's the control dependent on itself.

Similarly, we've got this self on B3 true, there's a path that all gets back to it, and there's a path that doesn't.

So, you can check yourself by going through the definitions. Okay?

So, all these are control-dependencies.

What we talked about data dependencies, and what's one data dependency here.

D is to find here, used here, so there's data dependency.

Before de-bugging the program, and we see something incorrect.

Say we are inspecting variable values and you see if the value D is wrong here.

And, look back and this is how to de-compute it.

This computer using B and D, I'm going to look at where they are.

I'm going to look at where they are dependent on.

Which happens to be this, and I think this. So, I am looking my way back.

And the data dependencies then are helping influence that computation.

But, so are the control-dependencies.

I can't use that one for this, but in here, the computation that happens in the node depends on this predict..on, how, there's no predicate to show here.

But whatever predicate is in there, that's causing these de-values that predicate, is determining what you get there in node.

And it's influencing your computation.

So, as a debugger if programs are using something wrong, as a person doing debugging, I'm interested in these control dependencies, as well.

Cause they are showing something about what drove the computation through.

So, let's see. Let's draw some of the dependencies. What data dependencies appear?

At first, is A.

So, from node one, definition of A to uses.

In node two, A to use a node four, A to use a node five, that's it.

B to use node two.

I'll just leave one edge for that.

Then each represents multiple dependencies. B is to we use a node five, we've

already got that. B to be used a node, no, B3. We already have five, too, B1 to B5. Oh, I did it again! B5. B5, that's B3. Did I get something wrong the, no, in A, it was B5. Okay.

B,B,B,B to node B3. That's what that is. And that is different. B to node B4, an, we already have that.

Ask me if I'm wrong. B1 to B5 does occur, also, and it gets killed.

Node two, we haven't think of C. the only use of C is going down two to five is the one.

It's going to get messy.

D as use in five, and it is killed and has used down to six.

Excuse me. It's used in three and six.

So, two goes to three and six.

D, oh, it's killed, you're right.

There's no path. Thank you. It only goes to five, doesn't it?

And was that alive when B3 to B5 to the C? Don't need to do many more of this, but the D gets killed, B5 gets down to D here gets down to here, three and six.

I'm going to stop there.

You can do the rest.

You can do the data-flow now, obviously, and get them accurate.

Or you're going to build a tool to get them accurate.

But if you want to know, the things that influence the node here, fruity bugger, if you are a person do de-bugging.

And you've noticed something going wrong in B4, right? One way, you could do this if you had that graph, use to trace the dependencies, the things that influence B4.

Well. B4 is control-dependent on B3, and data dependent on B1.

So I can go back far. B3 is data-dependent on B2, which is control-dependent on start.

B1 is control-dependent on start.

What else do we have? Ah, B3 back to B1. B2 back to B1.

So, I've ended up coloring, if I do this right.

So it hasn't excluded anything it has said these 3 influences it, ok?

That's called a slice of the program on it's on a slice in criteria just like these criteria might be these two definitions.

In the way I did it and the slice is the set of computations in the program that influence this note through chairs of control and data dependencies and one of the more famous papers in computer science in software engineering is by Marc Weiser.

I'd like to check this citation count on that paper.

His paper is entitled programmers use slices when debugging.

And it's actually, it's actually a little human study in there, where he shows when people are doing debugging activities they follow chains of control dependencies and then he said we can calculate those and he can offer them algorithm, now he didn't use the PDG to do it.

That was done later. It was actually earlier in his algorithm but this was the first paper, it was published in XC became the most influential paper in ten years for XC probably has thousands of hundreds of citations if not thousand I don't know we check sites here 3000 citations because anyone writes a paper on slicing and believe me there have been, well I don't know if there is 3000, there is a ton of papers on slicing.

First off there is slicing techniques many papers on those they had the site Wiser and now there is papers on uses of slicing like can we use slices when debugging can we use slices when doing Y? Can we use them when doing Z?

So this is where this gets. In this slicing now this slice is on a program dependence graph which represents a single procedure ok we are probably interested in the whole programs and for that we will need something little a deeper than a program dependence graph which will look at maybe in 3 classes.

All the system dependence graph. So probably one of the next most famous slicing paper is going to be the system dependence graph and its use in program slicing but that will be coming up so all this stuff came together we need to control photographs to do data floor our analysis and dominate analysis and a little bit of control dependence and we need the dominators do control dependence and there are other chains of influence there but these things all do lead to slicing which is a pretty important technique they also lead to other things.

Question about this?

Interesting that Waiser he did that paper and maybe one other and then he stopped working on the slicing went on and worked on I think he worked on ubiquitous computing and he died a few years ago but it's interesting to write a paper one other and get 3000 citations and then not keep you know most of us keep milking our papers.

For Oh what we do this, we can do that, we can do that that something we say in English milking our papers and we drain in the last ounce out of the work draining every bit we can but he went on ok.

I think we will stop a little early today 'cause I have papers due too and I think the next two classes will probably be talking about some basic testing technique well than probably the system dependence graph and then that takes us to choose sides right after that we are going to talk about projects. So..Ok.