Notes on Graph Algorithms Used in Optimizing Compilers

Carl D. Offner

University of Massachusetts Boston

April 26, 2011

Contents

1 Depth-First Walks **1**

1.1 Depth-First Walks . . . 1

1.2 A Characterization of DAGS . . . 8

1.3 A Characterization of Descendants . . . 10

1.4 The Path Lemma . . . 11

1.5 An Application: Strongly Connected Components . . . 11

1.6 Tarjan's Original Algorithm . . . 13

2 Flow Graphs **19**

2.1 Flow Graphs . . . 19

2.2 Dominators . . . 20

2.3 Depth-First Spanning Trees . . . 22

3 Reducible Flow Graphs **24**

3.1 Intervals . . . 24

3.2 Algorithms for Constructing Intervals . . . 27

3.3 Reducible Flow Graphs . . . 28

3.4 A Subgraph of Nonreducible Flow Graphs . . . 30

3.5 Back Arcs in Intervals . . . 33

3.6 Induced Depth-First Walks . . . 34

3.7 Characterizations of Reducibility . . . 36

3.8 The Loop-Connectedness Number . . . 39

3.9 Interval Nesting . . . 40

4 Testing for Reducibility **45**

4.1 Finite Church-Rosser Transformations . . . 45

4.2 The Transformations $T1$ and $T2$ . . . 46
4.3 Induced Walks from $T1$ and $T2$ . . . 49
4.4 Kernels of Intervals . . . 51
4.5 Reachunder Sets and Tarjan's Algorithm . . . 53

5 Applications to Data-Flow Analysis 59

5.1 Four Data-Flow Problems . . . 59
5.2 Data-Flow Equations . . . 61
5.3 Indeterminacy of Solutions . . . 65
5.4 Abstract Frameworks for Data-Flow Analysis . . . 66
5.5 Solutions of Abstract Data Flow Problems . . . 70
5.6 Two More Data-Flow Analysis Problems . . . 75
5.7 How Many Iterations are Needed? . . . 78
5.8 Algorithms Based on Reducibility . . . 82
5.8.1 Allen and Cocke's Algorithm . . . 83
5.8.2 Schwartz and Sharir's Algorithm . . . 88
5.8.3 Dealing With Non-Reducible Flow Graphs . . . 88

Chapter 1

Depth-First Walks

We will be working with directed graphs in this set of notes. Such graphs are used in compilers for modeling internal representations of programs being compiled, and also for modeling dependence graphs. In these notes, however, we will be concerned mainly with the graph theory; relations to compiler optimization will appear as applications of the theory.

All graphs in these notes are finite graphs. This fact may or may not be mentioned, but it should always be assumed.

The elements of a directed graph G are called nodes, points, or vertices. If x and y are nodes in G, an arc (edge) from x to y is written $x \to y$. We refer to x as the *source* or *tail* of the arc and to y as the *target* or *head* of the arc.[1]

To be precise, we should really denote a directed graph by $\langle G, A \rangle$, where A is the set of arcs (i.e. edges) in the graph. However, we will generally omit reference to A.

Similarly, a sub graph of a directed graph $\langle G, A \rangle$ should really be denoted $\langle H, E \rangle$, where E is a collection of edges in A connecting elements of H. However, in general we denote a sub graph of G simply by H. We will make the convention that the edges in the sub graph consist of *all* edges in the flow graph connecting members of H.

The one exception to this convention is when the sub graph H of G is a tree or a forest of trees. In this case, the edges of H are understood to be only the tree edges, ignoring any other edges of G between members of H.

1.1 Depth-First Walks

Algorithm A in Figure 1.1 performs a depth-first walk of a directed graph G and constructs

1. A depth-first spanning forest D of G.
2. A pre-order numbering of the nodes of G.

[1] While this usage is now standard, Tarjan's early papers use "head" and "tail" in the opposite sense.

3. A reverse post-order numbering of the nodes of G.

A depth-first walk is also often referred to as a depth-first *traversal*, or a depth-first *search*.

The numbering of the nodes of G by a depth-first walk is a powerful tool that can be used both to analyse structural properties of G and to understand the workings of graph algorithms on G. We will denote the pre-order number of a node x by pre[x], and the reverse post-order number of x by rpost[x].

```
procedure DFWalk(G: graph)
begin
    Mark all elements of G "unvisited";
    i ← 1;
    j ← number of elements of G;
    while there is an "unvisited" element x ∈ G do
        call DFW(x);
    end while;
end

procedure DFW(x: node)
begin
    Mark x "visited";
    pre[x] ← i; i ← i + 1;
    for each successor y of x do
        if y is "unvisited" then
            Add arc x → y to D;
            call DFW(y);
        end if;
    end for;
    rpost[x] ← j; j ← j − 1;
end
```

Figure 1.1: Algorithm A: Depth-First Walk and Numbering Algorithm

---

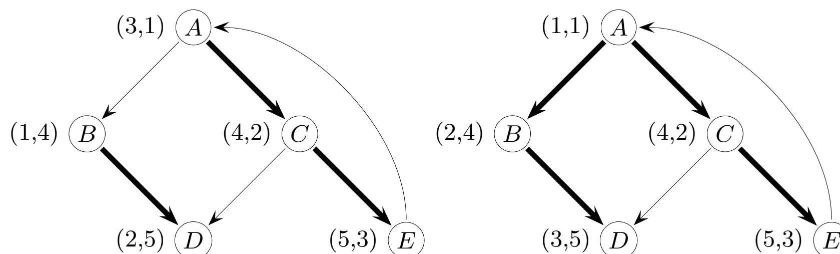
The successive calls to DFW from DFWalk yield a partition of G into sets of nodes, each with its own spanning tree. In particular, each call of DFW(x) from DFWalk yields a spanning tree of a set of noes of G. Different choices of x in the FOR loop of DFS will yield the same set of nodes of G but a different spanning tree for them.

Different choices of the next x to process in the WHILE loop of DFWalk will in general yield a different partition of G into sets of nodes.

The terms *successor* and *predecessor* always refer to the relation defined by the edges of the directed graph G.

On the other hand, the terms *ancestor*, *child*, and *descendant* are always to be understood with reference to a particular spanning forest D for G. So in particular, a node x may be an ancestor of another node y with respect to one spanning forest for G, but not with respect to another. Figure 1.2 shows two different walks of a graph G. The edges of the depth-first spanning forest in each case are indicated by thick arcs in

the graph.



Each node is labeled with an ordered pair of numbers (x, y), where x is the pre-order number of the node and y is the reverse post-order number of the node. The thick arcs are the arcs of the depth-first spanning forest.

Figure 1.2: Two depth-first forests for a directed graph.

If G is a tree, the pre-order and reverse post-order numberings are essentially equivalent; each node is visited before all its children. Thus, both pre-order and reverse post-order numberings of a tree topologically sort the partial order determined by the tree.

In the case of a DAG (directed acyclic graph), however, these orderings are not equivalent: the pre-ordering corresponds to a walk which visits a node not before *all* its children, but only before all those children which were previously unvisited. Thus, in a pre-order walk of a DAG, a node can be visited after one or more of its children. Reverse post-ordering does not suffer from this defect: it's probably intuitively obvious, but we will show precisely below (see Theorem 1.8 on page 9), in a post-order walk, a node is visited after all its children (whether the children were previously visited or not). Therefore, in a reverse post-order walk on a dag, a node comes before all its children, and therefore a reverse post-order numbering of a DAG topological sorts the DAG's partial order.

As an example, if in Figure 1.2, we remove the curved arc, the directed graph becomes a DAG. We see that, with respect to the first depth-first spanning forest (on the left), the pre-order numbering does not topologically sort the DAG's partial order, but the reverse post-order numbering does. (This is only an example, of course—as we've said, the actual proof for the general case of *any* DAG will come later.)

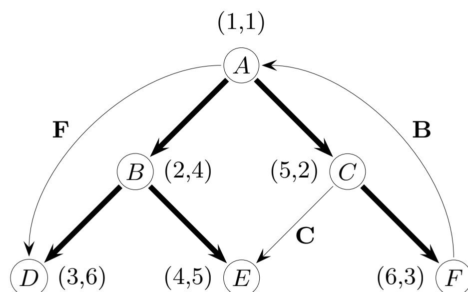
1.1 Exercise *Construct a DAG and a depth-first walk of the DAG such that the pre-order numbering corresponding to the walk does not topologically sort the DAG. Show that the reverse post-order numbering, however, does.*

The arcs in the depth-first spanning tree created by Algorithm A are all arcs that were originally in G. We call these arcs *tree arcs*. The remainder of the arcs in G can be classified as follows:

- Forward arcs: arcs that go from ancestors to descendants in D.
- Back arcs: arcs that go from descendants to ancestors in D, or from a node to itself.

- Cross arcs: all other arcs.[2]

Figure 1.3 illustrates a directed graph with arcs labelled as determined by a particular depth-first forest; in this case, the forest is actually a tree.



The ordered pairs are the pre- and post-order numbers of the nodes. The thick arcs are the arcs of the depth-first spanning tree (i.e. they are the tree arcs). The remaining arcs are labelled **F** (forward arc), **B** (back arc), and **C** (cross arc).

Figure 1.3: Classification of arcs in a depth-first walk.

As we remarked above, there are in general different depth-first spanning forests D of G. Corresponding to each such D, there is a different classification of arcs.

By examining the way Algorithm A works, we can write down some characterizations of these arcs. Let us distinguish between *reaching* a node and *processing* a node: a node may be reached several times in the algorithm (i.e., once for each predecessor), but is only processed once (the first time it is reached). The first time a node is reached is when a tree arc is created with that node as target. When a node x is processed, it is (figuratively, at least) placed on the stack by the recursion and pre[x] is assigned. When the processing is complete (i.e. after all its successors have been processed), it is taken off the stack and rpost[x] is assigned. Thus we see that node x is an ancestor of node y in D iff x is on the stack when y is first reached.

Another way to look at this is to modify Algorithm A so that instead of computing pre- and post-order numbers for each node, it computes the "times" start[x] and finish[x] that the node started and finished being processed; i.e. the times that it was pushed onto and popped from the stack. This algorithm is shown in Figure 1.4.

The relation of these time stamps to the numberings computed in Algorithm A is this:

- The numbers start[x] are just the pre-order numbers with gaps in them—every number finish[x] is a number missing from the sequence {start[x]}. Thus, if the sequence {start[x]} is "squashed" to remove

[2] The names for arcs given here differ from those used in some of the original papers:

Here	Tarjan[19]	Tarjan[20]
forward arc	reverse frond	forward arc
back arc	frond	cycle arc
cross arc	cross-link	cross arc

```
procedure DFWalk(G: graph)
begin
    Mark all elements of G "unvisited";
    i ← 1;
    while there is an "unvisited" element x ∈ G do
        call DFW(x);
    end while;
end

procedure DFW(x: node)
begin
    Mark x "visited";
    start[x] ← i; i ← i + 1;
    for each successor y of x do
        if y is "unvisited" then
            Add arc x → y to D;
            call DFW(y);
        end if;
    end for;
    finish[x] ← i; i ← i + 1;
end
```

Figure 1.4: Depth-First Walk with Time Stamps

the gaps, the number assigned to node x by the squashed sequence is pre$[x]$. In particular, the following statements are equivalent:

$$\begin{aligned}\text{pre}[x] &< \text{pre}[y]\\ \text{start}[x] &< \text{start}[y]\end{aligned}$$

x is pushed on the stack before y is.

- The numbers finish$[x]$ are just the post-order numbers with gaps in them. Squashing this sequence, as above, assigns to each node its post-order number. In particular, the following statements are equivalent:

$$\begin{aligned}\text{rpost}[x] &> \text{rpost}[y]\\ \text{finish}[x] &< \text{finish}[y]\end{aligned}$$

x is popped from the stack before y is.

In implementations, pre- and reverse post-order numbers are more useful than time stamps, because there are no gaps in the sequences of those numbers. Time stamps are useful, not so much in themselves, but because they make it somewhat easier to deduce properties of pre- and post-order numberings. This is

because, in addition to the relations just noted, they satisfy two additional properties which we state in the form of two lemmas:

1.2 Lemma *For each node x,* $\text{start}[x] < \text{finish}[x]$.

PROOF. This just says that x is always pushed on the stack *before* it is popped off the stack. □

1.3 Lemma (Parenthesis Nesting Property) *For any two nodes x and y, the start and finish numbers of x and y nest as parentheses. That is, only the following relations are possible (see Figure 1.5):*

$$\text{start}[x] < \text{finish}[x] < \text{start}[y] < \text{finish}[y]$$
$$\text{start}[x] < \text{start}[y] < \text{finish}[y] < \text{finish}[x]$$
$$\text{start}[y] < \text{start}[x] < \text{finish}[x] < \text{finish}[y]$$
$$\text{start}[y] < \text{finish}[y] < \text{start}[x] < \text{finish}[x]$$

PROOF. If $\text{start}[x] < \text{start}[y] < \text{finish}[x]$ then x is on the stack when y is first reached. Therefore the processing of y starts while x is on the stack, and so it also must finish while x is on the stack: we have $\text{start}[x] < \text{start}[y] < \text{finish}[y] < \text{finish}[x]$. The case when $\text{start}[y] < \text{start}[x] < \text{finish}[y]$ is handled in the same way. □

Another way to state the parenthesis nesting property is that given any two nodes x and y, the intervals $[\text{start}[x], \text{finish}[x]]$ and $[\text{start}[y], \text{finish}[y]]$ must be either nested or disjoint.

1.4 Lemma *x is an ancestor of y iff both* $\text{pre}[x] < \text{pre}[y]$ *and* $\text{rpost}[x] < \text{rpost}[y]$.

PROOF. x is an ancestor of y iff x is first reached before y is and the processing of y is complete before the processing of x is. This in turn is true iff $\text{start}[x] < \text{start}[y] < \text{finish}[y] < \text{finish}[x]$. □

All four of the relations illustrated in Figure 1.5 are possible. However, when there is an arc in the graph G from x to y, the last one can never occur. This is the content of the next theorem:

1.5 Theorem *The following table characterizes the pre- and reverse post-order numberings of arcs $x \to y$ in a directed graph G with respect to a depth-first walk:*

Tree Arc	$\text{pre}[x] < \text{pre}[y]$	$\text{rpost}[x] < \text{rpost}[y]$
Forward Arc	$\text{pre}[x] < \text{pre}[y]$	$\text{rpost}[x] < \text{rpost}[y]$
Back Arc	$\text{pre}[x] \geq \text{pre}[y]$	$\text{rpost}[x] \geq \text{rpost}[y]$
Cross Arc	$\text{pre}[x] > \text{pre}[y]$	$\text{rpost}[x] < \text{rpost}[y]$

PROOF. The first three lines of the table follow from Lemma 1.4. As for the last line of the table, there are two possibilities:

- The processing of y began before the processing of x began. That is, $\text{start}[y] < \text{start}[x]$. Since y is not an ancestor of x, then we must also have $\text{finish}[y] < \text{start}[x] (< \text{finish}[x])$, and this is just the last line of the table.

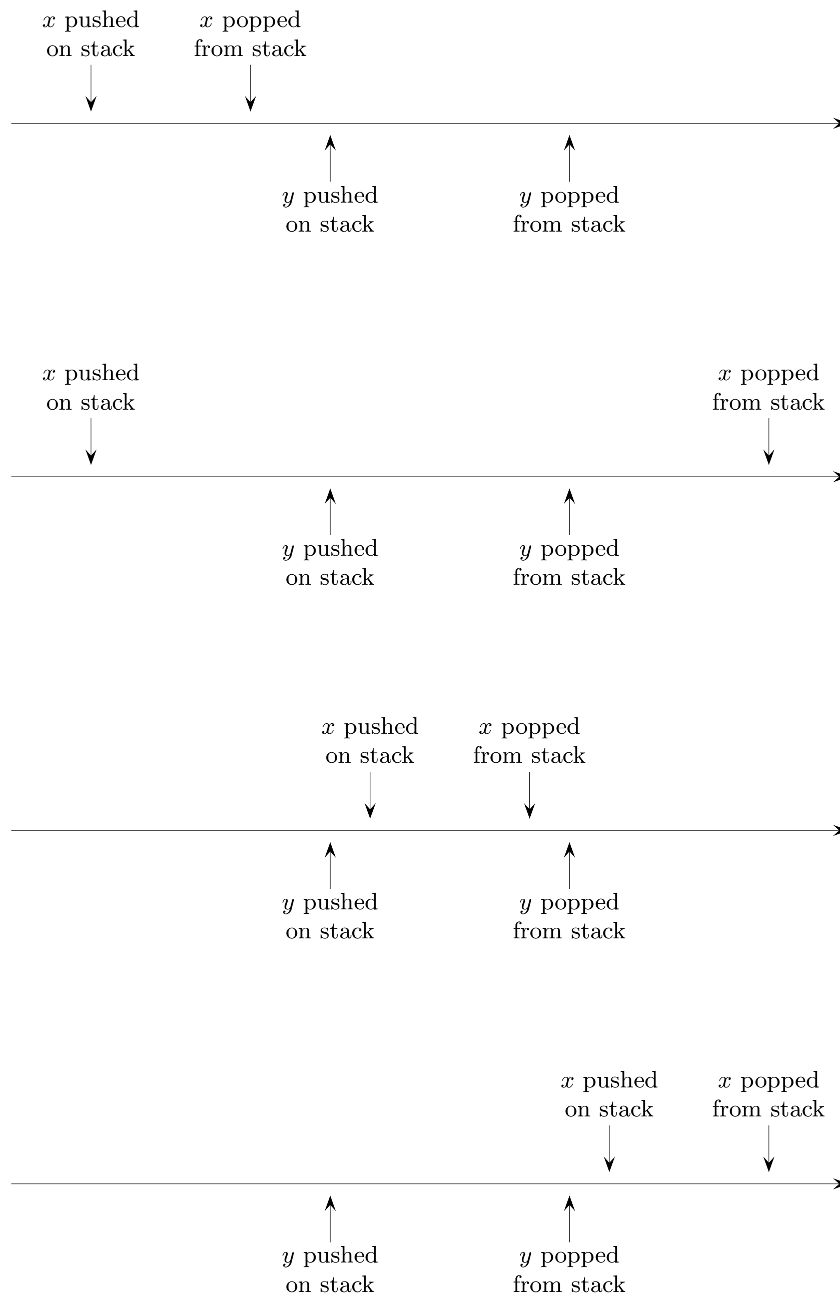


Figure 1.5: Parenthesis-like nesting of start-finish intervals for nodes x and y.

- y was not reached before the processing of x began. But in this case, since y is a successor of x, y will be reached while x is on the stack. That is, we have $\text{start}[x] < \text{start}[y] < \text{finish}[x]$, and so y is a descendant of x, contradicting the assumption that $x \to y$ is a cross arc. □

Following Tarjan[19], we modify Algorithm A by initializing both the arrays pre[ ] and rpost[ ] to 0. So at any point in the algorithm, $\text{pre}[x] = 0$ iff x has not yet been reached for the first time. (This does away with the need to mark nodes as "visited".) In addition, $\text{rpost}[x] = 0$ iff the processing of x is not finished. (It might not even have started, but at any rate, it's not finished.)

The modified algorithm, which we call Algorithm B (Figure 1.6), walks G and simultaneously constructs the pre-ordering and reverse post-ordering of G, classifies all the arcs in G (thereby constructing D as well), and computes the number of descendants $\text{ND}[x]$ of each node x in the tree D. (Note: we are using Tarjan's notation here for $\text{ND}[x]$. Actually, this value is the number of elements in the subtree of D which is rooted at x. Ordinarily, this would be thought of as 1 more than the number of descendants of x.)

1.6 Theorem *Algorithm B generates a spanning forest D of G, computes the number of descendants in D of each node x, and correctly classifies arcs in G relative to D.*

Proof. First, every node except the nodes x selected in the FOR loop is reached exactly once by a tree arc. So the tree arcs form a forest of trees rooted at those nodes x.

By the construction in the algorithm, if x is a terminal node in D then $\text{ND}[x] = 1$, and otherwise

$$\text{ND}[x] = 1 + \sum \{\text{ND}[y] : x \to y \text{ is an arc in } D\}$$

which is what we mean by the number of descendants of x.

Theorem 1.5 then shows that the rest of the arcs are labelled correctly. □

1.2 A Characterization of DAGS

Using these properties of arcs, we can characterize directed graphs which are DAGS:

1.7 Theorem *If G is a directed graph, the following are equivalent:*

1. *G is a DAG.*
2. *There is a depth-first walk of G with no back arcs. (More precisely, there is a depth-first walk of G with respect to which no arc in G is a back arc.)*
3. *No depth-first walk of G has back arcs. (More precisely, there is no arc in G which is a back arc with respect to any depth-first walk of G.)*

Proof. $1 \Longrightarrow 3$: Since there are no cycles in a DAG, there can be no back arcs with respect to any depth-first walk.

$3 \Longrightarrow 2$: because 3 is stronger than 2.

$2 \Longrightarrow 1$: By assumption, the nodes of G can be numbered by a depth-first walk of G with no associated back arcs. If G is not a DAG, then G contains a cycle. But then every edge $x \to y$ in the cycle would satisfy $\text{rpost}[x] < \text{rpost}[y]$, which is impossible. □

```
procedure DFWalk(G: graph)
begin
    i ← 1;
    j ← number of elements of G;
    for each node x ∈ G do
        pre[x] ← 0;
        rpost[x] ← 0;
    end for;
    while there is an element x ∈ G with pre[x] = 0 do
        call DFW(x);
    end while;
end

procedure DFW(x: node)
begin
    pre[x] ← i; i ← i + 1;
    ND[x] ← 1;    -- if x has no successors, ND[x] = 1
    for each successor y of x do
        if pre[y] = 0 then    -- y reached for the first time
            Label x → y a tree arc;    -- Add arc x → y to D
            call DFW(y);
            ND[x] ← ND[x] + ND[y];
        else if rpost[y] = 0 then    -- y is already on the stack
            Label x → y a back arc;
        else if pre[x] < pre[y] then    -- y's processing finished
            Label x → y a forward arc;
        else    -- y's processing finished
            Label x → y a cross arc;
        end if;
    end for;
    rpost[x] ← j; j ← j − 1;    -- x's processing finished
end
```

Figure 1.6: Algorithm B: Labelling and Numbering for a Depth-First Walk.

As a corollary, we can now see what we promised above:

1.8 Lemma *A reverse post-order numbering of a DAG topologically sorts the DAG's partial order.*

PROOF. Using the fact that in a depth-first walk of a DAG there are no back arcs, we see immediately from the table in Theorem 1.5 that the reverse post-order number of any node is less than that of each of its successors. And that immediately implies that the depth-first tree created by the walk is a topological sort of the DAG. □

1.3 A Characterization of Descendants

Again we let G be a directed graph with a depth-first walk which generates a spanning forest D by Algorithm B. We will characterize descendants in D. (Actually, we could just start with a spanning forest D and a depth-first walk of that forest, since the descendant relation is determined only by arcs in D.)

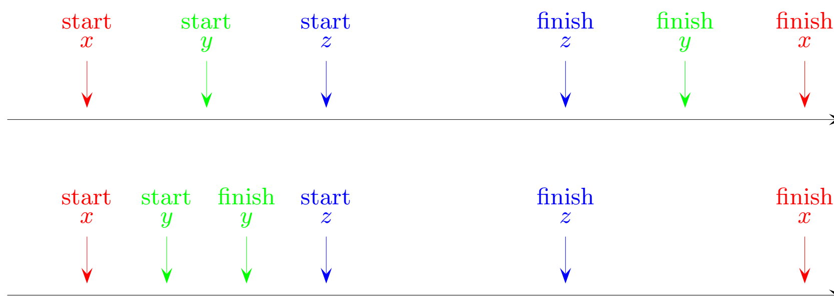
We already know by Lemma 1.4 that if z is a descendant of x, then $\mathrm{pre}[x] < \mathrm{pre}[z]$ and $\mathrm{rpost}[x] < \mathrm{rpost}[z]$.

1.9 Lemma

1. *If z is a descendant of x and if $\mathrm{pre}[x] < \mathrm{pre}[y] < \mathrm{pre}[z]$, then y is a descendant of x.*
2. *Similarly, if z is a descendant of x and if $\mathrm{rpost}[x] < \mathrm{rpost}[y] < \mathrm{rpost}[z]$, then y is a descendant of x.*

Proof. 1: This follows from the Parenthesis Nesting Property. Since z is a descendent of x, we must have the relations indicated in this picture:

Now since $\mathrm{pre}[x] < \mathrm{pre}[y] < \mathrm{pre}[z]$, we must have one of the two following situations:



and so y is a descendant of x.

2: The proof is similar. □

1.10 Lemma *The pre-order (resp. reverse post-order) numbers of all the descendants of a node x form an interval in $\mathbf{Z}$ whose left-hand endpoint is $\mathrm{pre}[x]$ (resp. $\mathrm{rpost}[x]$).*

Proof. We know that the pre-order numbers of all the descendants of x fall to the right of $\mathrm{pre}[x]$ in $\mathbf{Z}$. The preceding lemma then shows that there are no gaps in the set of these numbers. The proof for the reverse post-order numbers is similar. □

1.11 Theorem *The following are equivalent:*

1. *x is an ancestor of y.*
2. $\text{pre}[x] < \text{pre}[y]$ *and* $\text{rpost}[x] < \text{rpost}[y]$.
3. $\text{pre}[x] < \text{pre}[y] < \text{pre}[x] + \text{ND}[x]$.
4. $\text{rpost}[x] < \text{rpost}[y] < \text{rpost}[x] + \text{ND}[x]$.

PROOF. $1 \iff 2$ is just Lemma 1.4. $1 \iff 3$ and $1 \iff 4$ by the preceding lemma. □

1.4 The Path Lemma

1.12 Lemma *If there is a path $x = s_0 \to s_1 \to \ldots \to s_n = y$ such that* $\text{pre}[x] < \text{pre}[s_j]$ *for $1 \le j \le n$, then y is a descendant of x.*

Remarks 1. By the same token, every element of the path is a descendant of x.

2. The condition asserts that at the time that x is first reached, no element of the path from x to y has yet been reached.

PROOF. We will show by induction on j that each element s_j is a descendant of x. Of course this is true for $j = 0$. Now say it is true for a particular value j; i.e. say we know that s_j is a descendant of x. If $j = n$, of course, we are done. Otherwise, we have to show that s_{j+1} is also a descendant of x. To do this, we look at the arc $s_j \to s_{j+1}$.

If the arc is a tree arc or a forward arc, then s_{j+1} is a descendent of s_j, and we are done.

Otherwise, by Theorem 1.5, we have

$$\text{pre}[x] < \text{pre}[s_{j+1}] < \text{pre}[s_j]$$

so again s_{j+1} is a descendant of x, by Lemma 1.9. □

1.13 Lemma (Tarjan's "Path Lemma") *If* $\text{pre}[x] < \text{pre}[y]$*, then any path from x to y must contain a common ancestor of x and y.*

PROOF. This is an immediate corollary of Lemma 1.12: Let m be the node on the path such that $\text{pre}[s]$ is a minimum. By the Lemma, m is an ancestor of y. Further, either $m = x$ (in which case m is trivially an ancestor of x), or we have $\text{pre}[m] < \text{pre}[x] < \text{pre}[y]$, so m is also an ancestor of x by Lemma 1.9. □

Note that Lemma 1.12 also follows easily from Lemma 1.13.

1.5 An Application: Strongly Connected Components

In this section we present an efficient algorithm for computing the strongly connected components of an arbitrary directed graph G.

The *reverse graph* G^r of G is the graph with the same nodes as G but with each edge reversed (i.e. "pointing in the opposite direction"). It is immediate that the strongly connected components of G are the same as those of G^r.

1.14 Lemma *If $\{T_i\}$ is the spanning forest of trees produced by a depth-first traversal of G, then (the set of nodes of) each strongly connected component of G is contained in (the set of nodes of) one of the T_i.*

Remark Intuitively, once a depth-first walk reaches one node of a strongly connected component, it will reach all other nodes in that component in the same depth-first tree of the walk. The path lemma enables us to give a quick proof of this fact:

Proof. If x and y are contained in the same strongly connected component of G, let us suppose without loss of generality that $\text{pre}[x] < \text{pre}[y]$. Since we know there is a path from x to y, some element u of that path is an ancestor of x and y. That is, x, y, and u must all belong to the same tree in the forest. Since x and y are arbitrary, each element in the strongly connected component must belong to that same tree. □

---

procedure SCC(G: graph)
begin
  Perform a depth-first traversal of G, assigning to each node x its reverse-post-order number $\text{rpost}[x]$.
  Perform a depth-first traversal of G^r. Each time a choice is necessary in the **while** loop of the driving procedure DFWalk, choose the node x which has not yet been visited and for which $\text{rpost}[x]$ is least.
  Return the family of trees produced by this second walk. Each tree is a strongly connected component of G.
end

Figure 1.7: Algorithm for computing strongly-connected components.

1.15 Theorem *The algorithm in Figure 1.7 computes the strongly connected components of G.*

Proof. By the lemma, each strongly connected component lies entirely within some tree T in the forest produced by the second depth-first traversal (i.e., the traversal of G^r). So we only have to prove that each tree in the forest is strongly connected. To do this, it is enough to prove that if r is the root of a tree T in the forest and x is an element of T different from r, then x and r are in the same component; i.e. that there exist paths in G (or in G^r, for that matter) from x to r and from r to x. We may assume that $x \neq r$.

Since x is an element of the tree T rooted at r, there is a path in G^r from r to x (namely the path of tree arcs from r to x). The reverse path is therefore a path in G from x to r. We note that this path lies entirely in T^r.

Thus there is a path in G from x to r. It remains to show that there is a path in G from r to x.

To show this, we will show that in fact r is an ancestor of x in the original depth-first walk (the one over G in the algorithm).

Now r is an ancestor of x iff

$$\text{pre}[r] < \text{pre}[x]$$

$$\text{rpost}[r] < \text{rpost}[x]$$

Further, by the construction, we know that rpost$[r] <$ rpost$[x]$. (This is true because of the choice made in the second step of the algorithm.)

Therefore r is an ancestor of x iff pre$[r] <$ pre$[x]$.

Suppose then this is not true. Then (since $x \neq r$) pre$[x] <$ pre$[r]$, and so by the Path Lemma, there is an element u on the path in T^r from x to r which is an ancestor of both r and x. Hence rpost$[u] \leq$ rpost$[r]$, and since u is an ancestor of x (but by assumption r is not), $u \neq r$, so in fact rpost$[u] <$ rpost$[r]$, contradicting the choice of r made in the second step of the algorithm. □

This algorithm was published in Aho, Hopcroft, and Ullman[1], and is originally due to R. Kosaraju (unpublished) and Sharir[17].

Figure 1.8 gives an example of the workings of this algorithm.

1.6 Tarjan's Original Algorithm

Tarjan was the first person to give an algorithm for finding strongly connected components in time proportional to the size of G. Although this algorithm is more intricate than the algorithm presented in the previous section, it is quite easy to program, and is possibly somewhat more efficient.

We begin with a strengthening of Lemma 1.14:

1.16 Lemma *If T is any one of the trees generated by a depth-first search of G, then the intersection of the nodes* and edges *of T with those of a strongly connected component of G is again a tree.*

Proof. Let R be a strongly connected component of G. By Lemma 1.14, either R is disjoint from T or its set of nodes is completely contained in T. Further, from the proof of Lemma 1.14, we know that given any two elements x and y of R, they have a common ancestor in R. By iterating the process of finding common ancestors, we can thus find an element r of R which is a common ancestor of every other element in R.

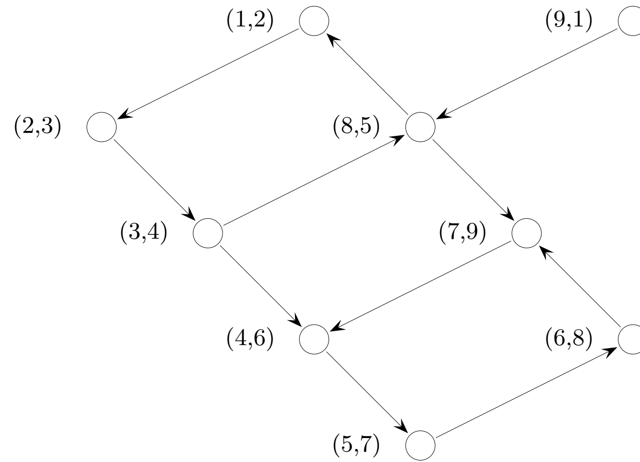
If y is any node in the component, and if $r \to^* x \to^* y$ is a path (x being any node in G) in T, then since there must be a path in G from y to r, x must also be in the component. Thus, the intersection of T with the component is itself a tree rooted at r. □

We refer to the node r as the *root* of the strongly connected component. r can be characterized as the node in the component whose pre-order number is least. Since it is an ancestor of every node in the component, its reverse post-order number is also least.

Note that r is not a unique entry node of R: a strongly connected component could have more than one entry node; and different depth-first searches of G could yield different roots of a strongly connected component.

Now for any node x in a strongly connected component R, consider the set of paths $\mathcal{P}(x)$ of the form $x = x_0 \to x_1 \to \ldots \to x_n \to y$ where

1. all the nodes in the path are in R,
2. all the arcs in the path except the last one are tree arcs (and so in particular, each arc is an arc from a node to a descendant of that node),



A depth-first walk in a directed graph. The ordered pairs by each node show the pre-order and reverse post-order numbers determined by the walk.

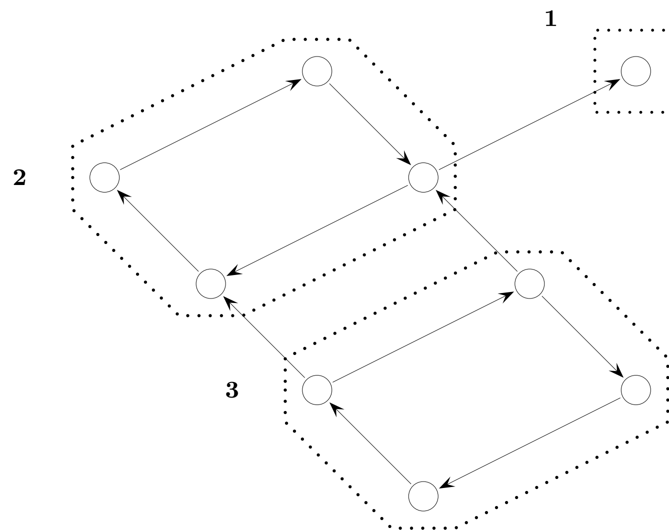


Figure 1.8: The reverse graph, showing the strongly connected components as found by the algorithm of Section 1.5. The numbers show the order in which the components are discovered.

3. the arc $x_n \to y$ is either a back arc or a cross arc.

Let $\mathcal{L}(x)$ denote the lesser of pre$[x]$ and the minimum value of pre$[y]$ where y is reachable from x by such an arc. So in particular, $\mathcal{L}(x) \leq \text{pre}[x]$.

Note that $\mathcal{L}(x)$ would have the same value if the arcs up to the last arc could also be forward arcs. Such a forward arc could always be replaced by a sequence of tree arcs. For our purposes, it is simpler to ignore the forward arcs.

1.17 Lemma *A node x is the root r of a strongly connected region* $\iff \mathcal{L}(x) = \text{pre}[x]$.

PROOF. $\Longrightarrow$: This is immediate from the fact that $\text{pre}[y] \geq \text{pre}[r]$ for any node y in R.

$\Longleftarrow$: Say $x \neq r$. Again, denote the depth-first spanning tree by T. By the previous lemma, $T \cap R$ is a tree T_R rooted at r. Since $x \neq r$, the subtree of T_R rooted at x (call it $T_{R,x}$) does not contain r. There is a path from x to r (and certainly every element of this path is in R). Let y be the first element of this path not in $T_{R,x}$. Then the predecessor p of y on this path can be reached by a series of arcs in $T_{R,x}$, and so that series of arcs followed by the arc $p \to y$ constitutes a path P in $\mathcal{P}(x)$.

Now if $\text{pre}[y] > \text{pre}[x]$, then by the Path Lemma, x would be an ancestor of y, which is impossible because $y \notin T_{R,x}$. Hence $\text{pre}[y] < \text{pre}[x]$, and so $\mathcal{L}(x) < x$. □

The problem with this Lemma is that without knowing R to begin with, condition 1 is impossible to evaluate. Therefore, we shall show below how to replace this condition with one that is easily deterimined. This is Tarjan's key point.

1.18 Lemma *If R is a strongly connected component of G with root r, and if $r \to x_1 \to x_2 \to \ldots \to x_n \to y$ is a path with all arcs except the last being either tree arcs or forward arcs, and if* $\text{pre}[y] < \text{pre}[r]$ *(so in particular, y does not lie in R), then if s is the root of the strongly connected component containing y, we have* $\text{rpost}[s] > \text{rpost}[r]$.

PROOF. An equivalent statement is this: if R and S are distinct strongly connected components with roots r and s respectively, and if $x \in R$ and $y \in S$ and there is an arc $x \to y$, and if $\text{pre}[y] < \text{pre}[r]$, then $\text{rpost}[s] > \text{rpost}[r]$.

We know in any case that $\text{pre}[s] \leq \text{pre}[y] < \text{pre}[r]$. If also $\text{rpost}[s] < \text{rpost}[r]$ then r would be a descendant of s. Since there is a path from r to s (containing the arc $x \to y$), this would mean that R and S are really the same strongly connected component, a contradiction. □

So here is how the algorithm works: We perform a depth-first search of the graph. We maintain an auxiliary stack σ of nodes of G. σ will be managed just like the implicit stack of the depth-first search, except that nodes are not automatically popped off it when their processing is finished.

To be precise, σ is managed as follows: As each node is first reached, it is pushed on σ. (That is, the nodes of G are pushed on σ in pre-order.) Thus the descendants of each node lie above the node on σ. In particular, the roots of the strongly connected components of G will eventually be on σ with the other elements of their component above them. When the processing (i.e. in the depth-first walk) for a root r is finished, we pop every element in the stack down to and including r. (For this to happen, of course, we have to be able to identify the nodes which are roots of strongly connected regions. We deal with this below.)

No other elements of other strongly connected regions can be among the elements popped. This is because if y were such an element, it would have been placed on the stack after r was first reached and before the processing of r was finished, and so would be a descendant of r. So by the preceding lemma, the root s of the component of y would also be a descendant of r, and so would have been placed above r on the stack. Since its processing would have been finished before r's processing, s and all nodes higher than it on the stack, including y, would already have been popped off the stack before we pop r.

As noted above, for this algorithm to work, we have to be able to identify the roots of the strongly connected regions. Using the stack σ, we can do this as follows: for each node x being processed, compute the attribute $L[x]$ (L is the replacement for $\mathcal{L}$ above), whose value is defined to be the lesser of $\mathrm{pre}[x]$ and the minimum of $\mathrm{pre}[y]$ taken over all y such that there is a path $x = x_0 \to x_1 \to \ldots \to x_n \to y$ where

1. Each arc up to x_n is a tree arc.
2. The arc $x_n \to y$ is a back arc or a cross arc.
3. y is on σ at some point during the processing of x.

Here condition 3 substitutes for the previous condition 1.

The same argument as in Lemma 1.17 shows that if x is not a root, then $L[x] < \mathrm{pre}[x]$.

On the other hand, let r be a root, and let y be that node for which $\mathrm{pre}[y] = L(r)$. Now the only way we could have $L[r] < \mathrm{pre}[r]$ is if the y belonged to a strongly connected component with root s such that $\mathrm{rpost}[s] > \mathrm{rpost}[r]$. Since we also have $\mathrm{pre}[s] \le \mathrm{pre}[y] = L(r) < \mathrm{pre}[r]$, we have

$$\begin{aligned} \mathrm{start}[s] &< \mathrm{start}[r] \\ \mathrm{finish}[s] &< \mathrm{finish}[r] \end{aligned}$$

and so by the parenthesis nesting property, $\mathrm{finish}[s] < \mathrm{start}[r]$; i.e., the processing of s is finished before the processing of r starts, which means that s (and hence y) is not on the stack at any time during the processing of r. Hence $L[r] = \mathrm{pre}[r]$.

Thus in the algorithm, we compute $L[x]$ for each node as it is processed. When the processing of the node is finished, if $L[x] = \mathrm{pre}[x]$, it is known to be a root, and it and all elements above it on the stack are popped and constitute a strongly connected component.

The complete algorithm is shown in Figure 1.9. Figure 1.10 shows the order in which the strongly connected components of the graph in Figure 1.8 are discovered, with the same depth-first walk as in that figure.

```
function SCCwalk(G: graph): set of sets of nodes
σ: stack of nodes;
SetOfComponents: set of sets of nodes;
begin
    Initialize the stack σ to be empty;
    SetOfComponents ← ∅;
    i ← 1;
    for each node x ∈ G do
        L[g] ← 0;
        pre[x] ← 0;
    end for;
    while there is an element x ∈ G with pre[x] = 0 do
        call SCC(x);
    end while;
    return SetOfComponents;
end

procedure SCC(x: node)
begin
    pre[x] ← i; i ← i + 1;
    L[x] ← pre[x];
    push x onto σ;
    for each successor y of x do
        if pre[y] = 0 then -- x → y is a tree arc.
            call SCC(y);
            if L[y] < L[x] then
                L[x] ← L[y];
            end if;
        else if pre[x] < pre[y] -- x → y is a forward arc.
            Do nothing;
        else -- x → y is a back or cross arc.
            if (y is on σ) and (pre[y] < L[x]) then
                L[x] ← pre[y];
            end if;
        end if;
    end for;
    if L[x] = pre[x] then
        C ← ∅;
        repeat
            pop z from σ;
            Add z to C;
        until z = x;
    end if;
    Add C to SetOfComponents;
end
```

Figure 1.9: Tarjan's original algorithm for finding strongly connected components.

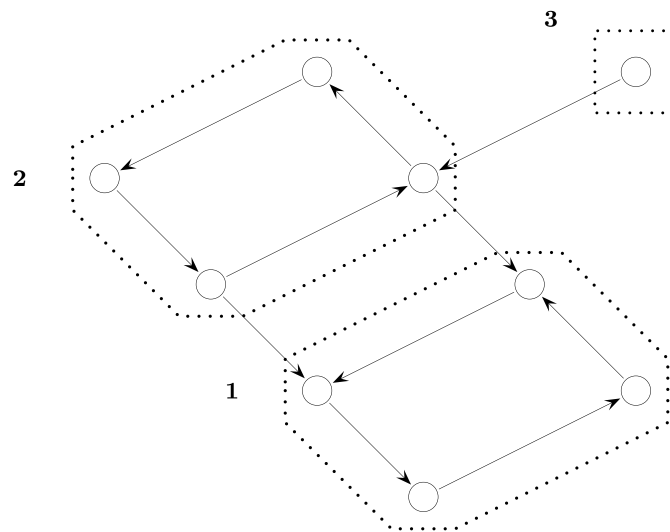


Figure 1.10: The same directed graph as in Figure 1.8. Given the same depth-first walk of this graph, this figure shows the order in which the, showing the strongly connected components are discovered by the algorithm of Section 1.6.

Chapter 2

Flow Graphs

2.1 Flow Graphs

A *flow graph* is a finite directed graph G together with a distinguished element s ("start", the "initial element") such that every element in G is reachable by a path from s. Henceforth, G will always be a flow graph with initial element s. As a notational convenience, we may sometimes write $\langle G, s\rangle$ to denote this flow graph.

A flow graph must have at least one node (i.e., s). A flow graph with only one node is called the *trivial flow graph*.

If P is a path $x_0 \to x_1 \to \ldots \to x_n$ in G, we call the nodes $\{x_1, \ldots, x_{n-1}\}$ the *interior* points of the path. A *loop* or *cycle* is a path which starts and ends at the same node. A *self-loop* is a loop of the form $x \to x$. We say that the path P is *simple* or *cycle-free* iff all the nodes $\{x_j : 0 \leq j \leq n\}$ in it are distinct.

If there is a path from x to y, then there is a *simple* path from x to y. (This is a simple proof by induction on the number of elements in the path. If the path is not simple, then there must be a cycle in it; removing the cycle is the inductive step.)

Most of our conventional usage for directed graphs applies in particular to flow graphs:

- If $x \to y$ is an arc in a flow graph, we say that x is a *predecessor* of y, and that y is a *successor* of x.
- The terms *ancestor*, *child*, and *descendant* have different meanings: they always refer to relations within a tree (e.g. a dominator tree, or the spanning tree of a flow graph).
- To be precise, we should really denote a flow graph by $\langle G, A, s\rangle$, where A is the set of arcs (i.e. edges) in the graph. However, we will generally omit reference to A.
- Similarly, a sub graph of a flow graph $\langle G, A, s\rangle$ should really be denoted $\langle H, E\rangle$, where E is a collection of edges in A connecting elements of H. However, in general we denote a sub graph of G simply by H. We will make the convention that the edges in the sub graph consist of *all* edges in the flow graph connecting members of H.

- The one exception to this convention is when the sub graph H of G is a tree or a forest of trees. In this case, the edges of H are understood to be only the tree edges, ignoring any other edges of G between members of H.

2.2 Dominators

If x and y are two elements in a flow graph G, then x *dominates* y (x is a *dominator* of y) iff every path from s to y includes x. If x dominates y, we will write $x \underline{\gg} y$. For example, in Figure 2.1 (modified from a graph in Tarjan[19]), C dominates J, but C does not dominate I.

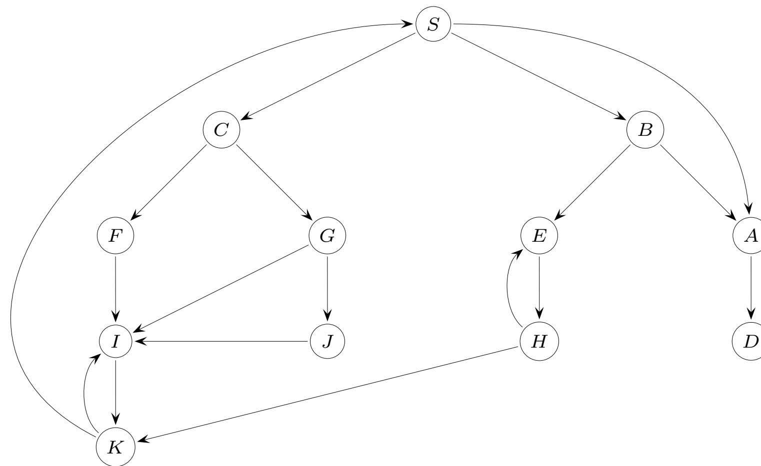


Figure 2.1: A flow graph in which we wish to find dominators.

In particular, if x is any element of G, then x dominates itself, and s dominates x. If $x \underline{\gg} y$ and $x \neq y$, we say that x *strongly dominates* y, and write $s \gg y$. The negation of $x \underline{\gg} y$ is $x \not\underline{\gg}\, y$, and the negation of $x \gg y$ is $x \not\gg y$.

2.1 Lemma *The dominance relation is a partial order on G. That is, for all x, y, and z in G,*

1. $x \underline{\gg} x$.
2. *If* $x \underline{\gg} y$ *and* $y \underline{\gg} x$, *then* $x = y$.
3. *If* $x \underline{\gg} y$ *and* $y \underline{\gg} z$, *then* $x \underline{\gg} z$.

PROOF. 1 and 3 are obvious.

To prove 2, let P be a simple path from s to y. Then P must include x. But then the initial part of the path from s to x must itself include y. So unless $x = y$, the path includes y in two different positions, and is not simple, a contradiction. □

2.2 Lemma *If x and y both dominate z then either $x \underline{\gg} y$ or $y \underline{\gg} x$.*

Proof. Let P be a simple path from s to z. P must contain both x and y. If y occurs after x, then $x \underline{\gg} y$, for if not, there would be a path P' from s to y which did not include x. Replacing that part of P from s to y with P' yields a path from s to z not including x, a contradiction. Similarly, if x occurs after y in P, then $y \underline{\gg} x$. □

Thus, the set of dominators of a node x is linearly ordered. In particular, every element x of G except s has an *immediate dominator*; i.e. an element y such that $y \gg x$ and if $z \gg x$ then $z \underline{\gg} y$. We denote the immediate dominator of x by $\text{idom}(x)$.

Now let us define a graph T on the elements of G as follows: there is an arc from x to y iff x is the immediate dominator of y.

2.3 Lemma *T is a tree rooted at s.*

Proof. s has no predecessors in T. (s has no dominators other than s itself.)

The in-degree of each element except s is 1. (There is only 1 arc leading to any $x \neq s$; namely the arc from its immediate dominator.)

Every element in G is reachable from s in T. (s dominates every element in G, and following the immediate dominator chain of any element x back from x leads to s.) □

T is called the *dominator tree* of G. Figure 2.2 shows the dominator tree for the flow graph of Figure 2.1.

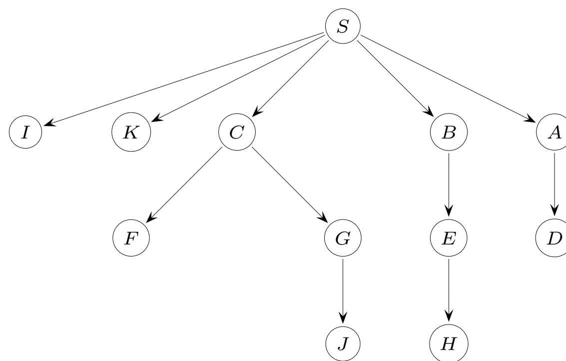


Figure 2.2: The dominator tree for the flow graph of Figure 2.1.

There are some relations between the arcs in $\langle G, s\rangle$ and the arcs in its dominator tree T. An arc in T may correspond to an arc in $\langle G, s\rangle$, or it may not. (For instance, the arc $S \to I$ in the dominator tree in Figure 2.2 does not correspond to an arc in the flow graph in Figure 2.1.) In general, an arc in $\langle G, s\rangle$ falls into one of the following three categories:

1. Arcs $x \to y$ where x is an ancestor of y in T (i.e. x dominates y). In this case, x must in fact be the parent of y in T (i.e. $x = \text{idom}(y)$), since otherwise there would be a path from s to x and then to y which avoided $\text{idom}(y)$ in T, a contradiction. (As we have seen, however, not every arc in T has to come from an arc in $\langle G, s\rangle$.)

2. Arcs $x \to y$ where y is an ancestor of x in T (i.e. where y dominates x . (These we may call "back arcs"; cf. the beginning of the proof of Theorem 3.25.)
3. All other arcs $x \to y$. y cannot be s, since then $y \underline{\gg} x$, which was handled by the previous case. Therefore, y has an immediate dominator $z = \text{idom}(y)$. Further, z must be an ancestor of x in T (i.e. $z = \text{idom}(y) \underline{\gg} x$), since otherwise there would be a path from s to x, and then to y, which avoided z, again a contradiction.

Figure 2.1 contains examples of each of these three types of arcs. In any case, we see that if $x \to y$ is an arc in G then either y is an ancestor of x in T or the parent of y in T is an ancestor of x in T. That is, either y dominates x or the immediate dominator of y dominates x. We will give a name to this result:

2.4 Lemma (Dominator Lemma) *If $x \to y$ is an arc in a flow graph G, then either*

1. *$y \underline{\gg} x$, or*
2. *$y \neq s$ and $\text{idom}(y) \underline{\gg} x$.*

Equivalently, either $y = s$ or $\text{idom}(y) \underline{\gg} x$.

2.3 Depth-First Spanning Trees

We will perform depth-first walks of flow graphs. All such walks will start at s; hence the spanning forests they generate will all be trees. Algorithm B of Chapter 1 (Figure 1.6) thus can be rewritten a little more simply as shown in Figure 2.3.

2.5 Theorem *The pre-order and reverse post-order numberings corresponding to any depth-first walk of a flow graph G each topologically sort the dominance partial ordering in G.*

Proof. If x dominates y, then x must be an ancestor of y in any depth-first walk of G. Hence $\text{pre}[x] < \text{pre}[y]$ and $\text{rpost}[x] < \text{rpost}[y]$ by the lemma. □

The study of flow graphs inevitably revolves about the study of loops—if there are no loops in a flow graph, the graph is a DAG, and in some sense the flow of control in the computer program it represents is trivial. So one of the main problems that has to be faced is how to get a handle on the looping structure of a flow graph. The simplest idea is undoubtedly to look at strongly connected subgraphs of the flow graph. In fact, it is evident that every flow graph can be uniquely decomposed into maximal strongly connected subgraphs. The problem with this is twofold:

1. If the graph is a DAG, every node in the graph is a separate maximal strongly connected subgraph. This is too much detail—we really don't care about all those nodes individually.
2. If the program contains a triply nested loop, for instance, all three of those loops will be contained in the same maximal strongly connected subgraph. This is too little detail. We want the decomposition to reflect the structure of loop inclusion.

We will see in chapter 3 how we can expose the looping structure of a flow graph by means of a decomposition into *intervals*.

```
procedure DFWalk(G: graph)
begin
    i ← 1;
    j ← number of elements of G;
    for each node x ∈ G do
        pre[x] ← 0;
        rpost[x] ← 0;
    end for;
    call DFW(s);
end

procedure DFW(x: node)
begin
    pre[x] ← i; i ← i + 1;
    ND[x] ← 1;    -- if x has no successors, ND[x] = 1
    for each successor y of x do
        if pre[y] = 0 then    -- y reached for the first time
            Label x → y a tree arc;    -- Add arc x → y to D
            call DFW(y);
            ND[x] ← ND[x] + ND[y];
        else if rpost[y] = 0 then    -- y is already on the stack
            Label x → y a back arc;
        else if pre[x] < pre[y] then    -- y's processing finished
            Label x → y a forward arc;
        else    -- y's processing finished
            Label x → y a cross arc;
        end if;
    end for;
    rpost[x] ← j; j ← j − 1;    -- x's processing finished
end
```

Figure 2.3: Algorithm B: Labelling and Numbering for a Depth-First Walk.

Chapter 3

Reducible Flow Graphs

The definitions that follow are all motivated by the goal of finding intrinsic characterizations of single-entry loops; that is, loops that can be entered at only one flow graph node. Such loops can of course be nested—programmers do this all the time. We need to find a way of representing this nesting in a way that can be analyzed efficiently.

3.1 Intervals

If H is a subset of nodes of a flow graph $\langle G, s\rangle$, then an element h of H is called an *entry node* of H iff either

1. $h = s$; or
2. there is an edge $x \to h$ in G with $x \notin H$.

A *region* in a flow graph is a sub graph H with a unique entry node h.

In particular, a region is a sub flow graph. We can write $\langle H, h\rangle$ to denote a region with entry h. We note that since H has a unique entry node, if $s \in H$, we must have $s = h$. The term "header" is also used to describe an entry node of a region.

A *pre-interval* in a flow graph is a region $\langle H, h\rangle$ such that every cycle in H includes h. That is, a pre-interval in a flow graph is a sub graph H satisfying the following properties:

1. H has a unique entry node h.
2. Every cycle in H includes h.

An *interval* in a flow graph is a maximal pre-interval.[1]

3.1 Lemma *Every element in a region $\langle H, h\rangle$ is reachable from h by a path in H.*

[1] In the literature, no distinction is customarily made between an interval and what here is called a pre-interval, although sometimes an interval is referred to as a "maximal interval". I have introduced the term pre-interval to make the distinction explicit. Pre-intervals are useful in proofs and in understanding the algorithms, but intervals are the objects of real interest.

Proof. If x is any element of H, then since $\langle G, s\rangle$ is a flow graph, there is a path

$$P : s = x_0 \to x_1 \to x_2 \to \ldots \to x_n = x$$

Let x_{j-1} be the last element of this path not in H. Then since $x_{j-1} \to x_j$ is an arc in G, x_j is an entry node of H and so must must equal h. Thus the path

$$x_j \to \ldots \to x_n = x$$

is a path in H from h to x. □

3.2 Lemma *If $\langle H, h\rangle$ is a region and $x \in H$, any simple path from h to x lies entirely within H.*

Proof. If the path $h = x_0 \to x_1 \to \ldots \to x_n = x$ does not lie entirely within H, then as in the proof of the previous lemma, one of the points x_j with $j \geq 2$ is an entry node of H, and hence must be h. But then the path is not simple. □

3.3 Lemma *If $\langle H, h\rangle$ is a region, then h dominates (in G) every other node in H.*

Proof. If $s \in H$, then $h = s$, and so h dominates every element in G and so also in H.

Otherwise, there is a path from s to some node x in H which avoids h and which is not entirely contained in H. Let y be the last element on this path which is not in H, and let z be the next element on this path (so $z \in H$; z might be x, in particular). Then as before, z is an entry node of H but $z \neq h$, a contradiction. □

3.4 Theorem *For each element $h \in G$, there is a region $R(h)$ with header h which includes all other regions with header h. $R(h)$ consists of all elements of G which are dominated by h.*

Remark The region is not a "maximal region" unless $h = s$; in fact, the only maximal region is $\langle G, s\rangle$ itself. But the region *is* maximal among all regions with header h.

Proof. Let $\mathcal{H}$ denote the family of regions with header h. Since $\{h\} \in \mathcal{H}$, $\mathcal{H} \neq \emptyset$. The union of any two members of $\mathcal{H}$ is in $\mathcal{H}$. But then since $\mathcal{H}$ is finite, the union of all the members of $\mathcal{H}$ is in $\mathcal{H}$. Thus $R(h)$ exists.

Let $H = \{x : h \ggeq x\}$. By Lemma 3.3, $R(h) \subset H$.

To prove the converse inclusion, we will show that H is a region with header h—this will show that $H \subset R(h)$.

If H is not such a region, then it has an entry node x different from h. Therefore there is an arc $y \to x$ with $y \notin H$. Since then y is not dominated by h, there is a path P from s to y which avoids h. But then appending the arc $y \to x$ to the end of P yields a path from s to x which avoids h, a contradiction, because h dominates x. Thus H has h as its unique entry node, and is therefore a region, and so $H \subset R(h)$. □

3.5 Lemma *A region is a convex subset of the dominator tree. That is, if $\langle H, h\rangle$ is a region, if x and y are in H, and if there is an element $m \in G$ such that $x \ggeq m \ggeq y$, then $m \in H$.*

Proof. If P is a simple path from s to y, then since $h \ggeq x$, that part of P which starts at h is a simple path from h to y which includes x and m. By Lemma 3.2, it lies entirely within H, so $m \in H$. □

Remark Not every convex subset of the dominator tree is a region, however. For instance, the union of two disjoint subtrees of the dominator tree is not a region.

3.6 Lemma *If $\langle H, h\rangle$ and $\langle K, k\rangle$ are regions with a non-empty intersection, then either $k \in H$ or $h \in K$. If $k \in H$, then $\langle H \cup K, h\rangle$ is a region.*

PROOF. There must be a node $x \in H \cap K$. Since h and k both dominate x, either h dominates k or k dominates h. Say h dominates k. Then by Lemma 3.4, $k \in H$. But then $H \cup K$ must be a region with header h, for any path entering $H \cup K$ either enters H (at h) or K (at k). But since $k \in H$, to reach k it must first enter H at h. □

3.7 Lemma *If $\langle H, h\rangle$ and $\langle K, k\rangle$ are pre-intervals with a non-empty intersection, then either $k \in H$ or $h \in K$. If $k \in H$, then $\langle H \cup K, h\rangle$ is a pre-interval.*

PROOF. By the previous lemma, we already know that either $k \in H$ or $h \in K$ and that $H \cup K$ is a region. Without loss of generality, let us assume that $k \in H$. We will show that $\langle H \cup K, h\rangle$ is a pre-interval.

Let $C = \{x_0 \to x_1 \to \ldots \to x_n = x_0\}$ be a cycle in $H \cup K$. If $C \subset H$, then C must include h. Otherwise, there is a node $x_j \in K - H$, and there are two possibilities:

Case I: $C \subset K$**.** Then C includes k, which is in H, and so C enters H (since $x_j \notin H$), so C must enter H at h; i.e. $h \in C$.

Case II: $C \not\subset K$**.** Again, C must re-enter H (since it is not completely contained in K), so it must contain h.

So in any case, $h \in C$, and so $\langle H \cup K, h\rangle$ is a pre-interval. □

3.8 Theorem *For each element $h \in G$, there is a pre-interval with header h which includes all other pre-intervals with header h.*

Remark The pre-interval is not necessarily a maximal pre-interval (i.e. is not necessarily an interval). But the pre-interval *is* maximal among all pre-intervals with header h.

PROOF. As in the proof of Lemma 3.4, let $\mathcal{H}$ denote the family of pre-interals with header h. Since $\{h\} \in \mathcal{H}$, $\mathcal{H} \neq \emptyset$. Lemma 3.7 shows that the union of any two members of $\mathcal{H}$ is in $\mathcal{H}$. But then since $\mathcal{H}$ is finite, the union of all members of $\mathcal{H}$ is in $\mathcal{H}$. □

3.9 Theorem *If $\langle H, h\rangle$ and $\langle K, k\rangle$ are intervals in the flow graph $\langle G, s\rangle$, then they are either identical or disjoint.*

PROOF. If H and K are not disjoint, then by Lemma 3.7, $H \cup K$ is a pre-interval. But then by maximality, $H = H \cup K = K$. □

3.10 Theorem *G can be uniquely partitioned into (disjoint) intervals.*

PROOF. Any single node in G by itself constitutes a pre-interval. Hence every node in G lies in some interval. That interval is unique by the previous theorem, and the set of all these intervals is disjoint by the theorem. □

This last theorem is an essential and key fact concerning intervals.

```
function MakePI(h: node)
begin
    A ← {h};    -- initialize
    while there exists a node m ≠ s all of whose predecessors are in A do
        A ← A ∪ {m};
    end while;
    return A;
end
```

Figure 3.1: Algorithm C
Compute the Largest Pre-Interval with Header h

3.2 Algorithms for Constructing Intervals

For each $h \in G$, let $PI(h)$ be the largest pre-interval with header h. By Theorem 3.8, $PI(h)$ exists and contains every pre-interval with header h. We will show that Algorithm C in Figure 3.1 computes $PI(h)$:

3.11 Theorem *Algorithm C computes $PI(h)$.*

Proof. 1. Since all the predecessors of each node m which is added to A all lie in A already, by induction A has h as its unique entry point.

2. If there is a loop $x_0 \to x_1 \to \ldots \to x_n = x_0$ in A which avoids h, let x_j be the last element in this loop which was added to A by the algorithm. Then x_{j+1} must have already been in A. But this is a contradiction, for since $x_{j+1} \neq s$, it must have been added to A by the algorithm, and when this happened, one of its predecessors would have been x_j which was not yet in A.

Thus A is a pre-interval with header h. We must show that it is the maximal such pre-interval. So say B is a strictly larger pre-interval with header h. By the construction in the algorithm, every element of $B - A$ must have a predecessor in $B - A$; otherwise it would have been added to A. (Note that s cannot be in $B - A$, since B has header h.) But then, since B is finite, if we take any element $x \in B - A$ and find a predecessor in $B - A$, and iterate this process, we must eventually create a cycle in $B - A$. This cycle does not contain h, which contradicts the assumption that B is a pre-interval with header h. □

Now Algorithm D in Figure 3.2 can be used to partition G into intervals. The algorithm uses an auxiliary set H (a worklist of headers) and builds a family $\mathcal{J}$ of intervals.

3.12 Theorem *Algorithm D partitions G into intervals.*

Proof. Since all elements of G are reachable from s, the algorithm at least partitions G into pre-intervals. So we only have to show that each pre-interval created by the algorithm is an interval. We will proceed by induction.

The first pre-interval created is $PI(s)$. This must be an interval, because any larger pre-interval would also have header s; but $PI(s)$ includes all pre-intervals with header s.

Now assume that we are about to add $PI(h)$ to $\mathcal{J}$; by the inductive assumption all the pre-intervals already added to $\mathcal{J}$ were actually intervals. We must show that $PI(h)$ is also an interval.

```
procedure MakeIntervals
begin
    -- Initialize H and 𝒥:
    H ← {s};
    𝒥 ← ∅;
    Mark all nodes of G "unselected".
    while H ≠ ∅ do
        Extract a node h from H;
        Compute PI(h) by algorithm C;
        Mark all nodes of PI(h) "selected";
        Add PI(h) to 𝒥;
        Add to H any node m ∈ G which is not selected but which has a selected predecessor;
    end while;
    return 𝒥;
end
```

Figure 3.2: Algorithm D
Partition a Flow Graph into Intervals

In any case, $PI(h)$ is contained in an interval H, say. Since intervals are disjoint, H is disjoint from all the intervals in $\mathcal{J}$. Since h has a predecessor in one of the intervals in $\mathcal{J}$, h must be the unique entry point (i.e. the header) of H. But we know that $PI(h)$ contains every pre-interval with header h, so $PI(h) = H$. Hence $PI(h)$ is an interval, and we are done. □

If $PI(h)$ is actually an interval, we shall refer to it also as $I(h)$.

3.3 Reducible Flow Graphs

Corresponding to the flow graph G, we can create a *derived flow graph* $I(G)$ as follows:

1. The nodes of $I(G)$ are the intervals of G.
2. The initial node of $I(G)$ is $PI(s)$.
3. If I and J are two distinct intervals of G, there is an arc $I \to J$ in $I(G)$ iff there is an arc in G from some element of I to the header of J. (In particular, $I(G)$ contains no self-loops.)

Thus, each node in G has an image node in $I(G)$ (namely, the interval to which it belongs), and each path in G has an image path in $I(G)$. In general, each subgraph H of G has an image, which we will denote by $I(H)$, and which is a subgraph of $I(G)$. Going in the other direction, every path in $I(G)$ is the image of at least one path in G.

Note that the image of a simple path may not be simple. (A path can start in an interval but not at its header, leave that interval, and end by reentering the interval at its header. The path could be simple but the image would not be.)

On the other hand, every path in $I(G)$ is the image of a path in G, and a simple path in $I(G)$ can be taken to be the image of a simple path in G.

Now this process can be repeated: If we set $G_0 = G$ and for all $j > 0$ we set $G_j = I(G_{j-1})$, then the sequence $G_0, G_1, \ldots$ is called the *derived sequence* for G.

The number of elements of $I(G_j)$ is always $\leq$ the number of elements of G_j. Further, $I(G_j)$ has the same number of elements as G_j iff each element of G_j constitutes an interval of G_j, in which case $I(G_j) = G_j$. Thus, after a finite number of terms, each term of the derived sequence is a fixed flow graph $\hat{I}(G)$ called the *limit flow graph* of G. Every node of $\hat{I}(G)$ is an interval.

G is said to be *reducible* iff $\hat{I}(G)$ is the trivial flow graph (i.e. the flow graph with only one node). A flow graph which is not reducible is said to be *nonreducible*. Figure 3.3 shows the sequence of reductions of a reducible flow graph.

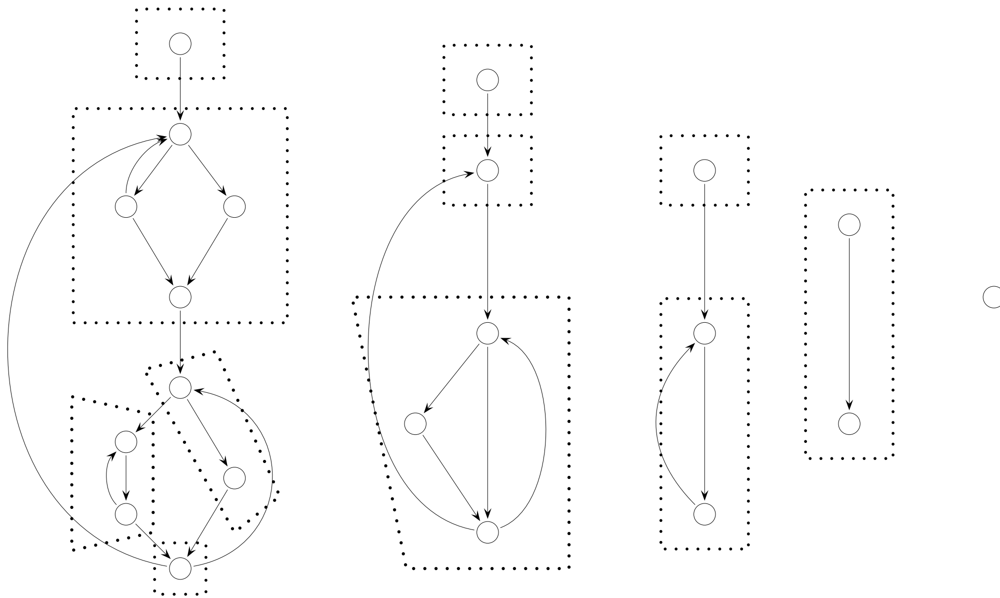


Figure 3.3: Derived Sequence of a Reducible Flow Graph

For example, if G is a directed acyclic graph (DAG), then G is reducible: since there are no cycles, $\langle G, s \rangle$ is itself an interval, so $I(G)$ already is the trivial flow graph.

We will say that a non-trivial limit flow graph is an *irreducible* flow graph. (Note, however, that in the literature, the terms *irreducible* and *nonreducible* have sometimes been used interchangeably.)

Flow graphs which are generated by programs written in languages with structured flow of control statements such as *if-then-else* constructions and *while* loops but no *goto* statements are automatically reducible. And in fact, it has been observed that competent programmers who do use *goto* statements virtually always produce code which leads to reducible flow graphs anyway. So nonreducible flow graphs really are in some sense pathological.

The concept of reducibility seems to be an intrinsically important one, as there are many striking characterizations of reducible flow graphs, as we will see below. Right now, we will prove a useful result relating the derived sequence to the dominance partial order:

3.13 Lemma *If $\langle H, h\rangle$ and $\langle K, k\rangle$ are intervals in G and if $x \in K$, the following are equivalent:*

1. $H \underline{\gg} K$ *in* $I(G)$.
2. $h \underline{\gg} k$ *in* G.
3. $h \underline{\gg} x$ *in* G.

Proof. $1 \Longrightarrow 3$: Otherwise, there would be a path from s to x which avoids h. But then this path cannot include any member of H, and so its image in $I(G)$ avoids H; hence H does not dominate K in $I(G)$.

$3 \Longrightarrow 2$: Otherwise there would be a path from s to k not including h. Adjoining to this path the path in K from k to x gives a path from s to x not including h, a contradiction.

$2 \Longrightarrow 1$: Any path from s to k must go through h, which means that any path in $I(G)$ from $I(s)$ to K must go through H, so $H \underline{\gg} K$ in $I(G)$. □

3.14 Theorem *If $x \to y$ is an arc in G, and if the images of x and y in an element G_j of the derived sequence of G are x_j and y_j respectively and if $x_j \neq y_j$, then $y \underline{\gg} x \iff y_j \underline{\gg} x_j$.*

Proof. Let us say the images of x and y in the elements G_k of the derived sequence of G are x_k and y_k respectively.

Since $x_j \neq y_j$, we must have $y_k \neq x_k$ for $0 \le k \le j$. Therefore the arc $x \to y$ has an image arc $x_k \to y_k$ in G_k for $0 \le k \le j$. For $0 \le k < j$, x_k and y_k cannot be in the same interval of G_k, and hence y_k must be an interval header in G_k. (This is where we use the fact that $x \to y$ is an arc.) Hence applying Lemma 3.13 repeatedly, we get

$$y \underline{\gg} x \iff y_1 \underline{\gg} x_1 \iff \ldots \iff y_j \underline{\gg} x_j.$$

□

3.4 A Subgraph of Nonreducible Flow Graphs

We will show that a flow graph is nonreducible iff it contains a subgraph of the form in Figure 3.4, where

1. the paths P, Q, R, S, and T are all simple.
2. the paths P, Q, R, S, and T have pairwise disjoint interiors.
3. nodes s, a, b, and c are distinct, unless $a = s$, in which case there is no path S in the subgraph.

4. nodes s, a, b, and c do not occur in the interiors of any of the paths P, Q, R, S, and T.

Let us call this subgraph $(*)$.

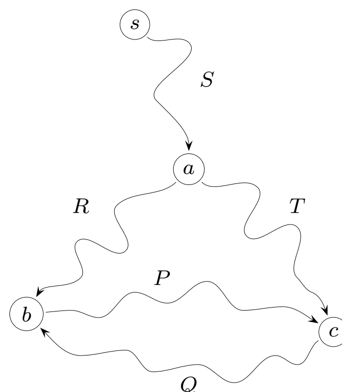


Figure 3.4: A Subgraph of Nonreducible Flow Graphs.

3.15 Lemma *If two paths in G have disjoint interiors, then their images in $I(G)$ have disjoint interiors.*

PROOF. If not, then both paths in G have an interior node in the same interval in G, and furthermore, the starting node of each path cannot be in that interval (for otherwise, that interval would not be an interior point of the corresponding image path). But then each path must have entered that interval at an interior point of the path, which means that the header node of the interval must be an interior point of each path, a contradiction. □

3.16 Lemma *If $(*)$ is a subgraph of $I(G)$, then $(*)$ is a subgraph of G.*

PROOF. For a, b, and c in G, take the headers of the (three distinct) intervals in G which are the vertices of $(*)$ in $I(G)$. The paths in G will be composed of arcs between intervals (which correspond to the arcs in the paths of $I(G)$), and arcs totally within intervals. Since no interval in G which corresponds to a node on any of the paths of $(*)$ in $I(G)$ occurs twice, these paths will all have disjoint interiors, and the nodes s, a, b, and c will not occur on the interior of any of these paths. Further, the paths can all be taken to be simple. □

3.17 Lemma *If $(*)$ is a subgraph of G, then $(*)$ is a subgraph of $I(G)$.*

PROOF. b and c cannot be in the same interval, since if they were, the paths P and Q would form a cycle which would have to include the header of that interval (either because it was completely contained within the interval or because it left the interval and re-entered it). But the path SR also enters the interval, and by assumption, $S \cup R$ intersects the cycle $P \cup Q$ only at b, so b would have to be the header of the interval. But similarly, c would have to be the header of the interval, and $b \neq c$, a contradiction.

a and b cannot be in the same interval, since if they were, the header of that interval would have to lie in S. (If $a = s$, the header of course would be a.) But then the cycle PQ could not be contained in that interval, since it does not contain the header, and it could not leave and re-enter the interval for the same reason.

Similarly, a and c cannot be in the same interval. Therefore, a, b, and c have distinct images in $I(G)$. Lemma 3.15 then shows that the images of P, Q, R, and T have disjoint interiors.

The image of a cannot lie on the interior of the image of S, for then S would have to enter the interval containing a twice and so would not be simple.

The image of a cannot lie on the interior of the image of R, for then the path SR would have to enter the interval containing a twice, which would again be a contradiction. Similarly, the image of a cannot lie on the interior of the image of T.

The image of a cannot lie on the interior of the image of P, for then the path SRP would enter the interval containing a twice, again a contradiction. And similarly, the image of a cannot lie on the interior of the image of Q.

Similar arguments show that the images of b and c cannot lie on the interior of the images of S, R, T, P, or Q.

And finally, the images of all the paths are simple, by analogous arguments. □

3.18 Theorem *A flow graph $\langle G, s\rangle$ is reducible iff it does not contain a subgraph $(*)$.*

PROOF. If G contains $(*)$ as a subgraph, then by Lemma 3.17, so does $I(G)$. Iterating, the limit $\hat{I}(G)$ must contain $(*)$ as a subgraph, which means it is not the trivial flow graph, so G is nonreducible.

Conversely, if G is nonreducible, then by Lemma 3.16, it is enough to show that $I(G)$ contains $(*)$. So we may assume that in fact, G is a limit flow graph. We will prove G contains $(*)$ by induction on the number of nodes of G.

First, any flow graph with 1 or 2 nodes is reducible. For flow graphs of 3 nodes, a simple enumeration of possibilities shows that the nonreducible ones contain $(*)$. So let us assume that it has been shown that for $3 \le j \le n-1$, any limit flow graph with j nodes contains $(*)$, and say G has n nodes. In particular, since G is a limit flow graph, every node in G is an interval.

With any depth-first spanning tree D for G, with its associated reverse post-order numbering of nodes of G, let x be the child of s whose reverse post-order number is smallest. (In fact, it must be 2.) There must be an arc $y \to x$ with $y \ne s$, since otherwise Algorithm C would show that x was in the interval containing s. Since $y \ne s$ and $y \ne x$, we must have $\mathrm{rpost}[y] \ge 2 = \mathrm{rpost}[x]$, so $y \to x$ is a back arc, and so x is an ancestor of y in D. Thus, there is a simple path A in D from x to y. We consider two cases:

1. x dominates y: Then $R(x)$ (see Theorem 3.4 on page 25) is a region including y but not including s. Any interval in this region (considered as a sub flow graph) must be a pre-interval in G. Hence the only intervals in $R(x)$ are single nodes: $R(x)$ considered as a sub flow graph is a limit flow graph with fewer than n nodes, and hence by induction contains $(*)$ with x corresponding to s or a. Prepending the arc $s \to x$ yields a copy of $(*)$ in G.
2. x does not dominate y: Then there is a simple path B from s to y not containing x. Let z be the first point on B (starting from s) which is also on A. z could be y, but it cannot be x. In either case, $\{s, z, x\}$ forms an example of $(*)$ in G. □

3.5 Back Arcs in Intervals

3.19 Lemma *if $\langle H, h\rangle$ is a pre-interval, and if x and y are nodes in H and x is an ancestor of y with respect to some depth-first walk of G, then there is a path P in H from x to y such that $h \notin P$ unless $x = h$.*

PROOF. By assumption, the depth-first spanning tree D of G which is created by the depth-first walk contains a simple path from s to x to y. This path must enter H at h, so h precedes x in P, and so that part of the path from x to y can contain h only if $x = h$. □

3.20 Theorem *If $\langle H, h\rangle$ is an interval, than an arc $x \to y$ between nodes of H is a back arc with respect to a depth-first spanning tree iff $y = h$.*

Remark Thus, the back arcs in an interval do not depend on which depth-first walk we use to construct a spanning tree of that interval. We will see below that this property is true of a flow graph as a whole iff the flow graph is reducible.

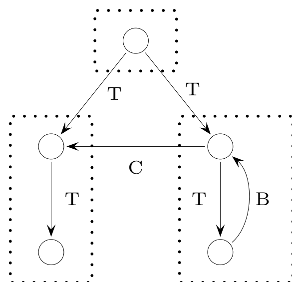
PROOF. If the arc were a back arc but $y \neq h$, then by Lemma 3.19, there would be a path P from y to x in H avoiding h. This path together with the back arc would constitute a cycle in H avoiding h, which contradicts the fact that $\langle H, h\rangle$ is a pre-interval.

Conversely, if $x \to h$ is an arc, then since h dominates x, h is an ancestor of x, and so the arc is a back arc. □

3.21 Corollary *The target of any back arc is an interval header.*

PROOF. If the arc is contained in an interval, the result follows from the theorem. Otherwise, the target of the arc must be an entry point of an interval, and so is the interval header. □

It is not true that every interval header is the target of a back arc. Figure 3.5, for instance, gives a counterexample.



Arcs are labeled T (tree arc), C (cross arc), and B (back arc). Intervals are indicated by dotted boxes. There is no back arc to the header of the left bottom interval.

Figure 3.5: An Interval Which is not the Target of a Back Arc.

3.6 Induced Depth-First Walks

3.22 Lemma *If D is a spanning tree for G, then $I(D)$ is a spanning tree for $I(G)$.*

Proof. If $\langle H, h\rangle$, $\langle I, k\rangle$, and $\langle K, k\rangle$ are three distinct intervals in G and $H \to K$ and $I \to K$ are arcs in $I(G)$ which come from arcs in D, then there must be arcs $x \to k$ and $y \to k$ in D with $x \in H$ and $y \in I$. Since H and I are disjoint, k has two parents in D, which is impossible, since D is a tree. Hence $I(D)$ is a tree, and it spans $I(G)$ because it enters every interval in G. □

We see that if $\langle H, h\rangle$ and $\langle K, k\rangle$ are intervals in G (i.e. nodes of $I(G)$), and if $H \to K$ is an arc in $I(G)$, then $H \to K$ is an arc in $I(D)$ iff there is a tree arc $x \to k$ with $x \in H$.

Now if we have a depth-first walk of G, there is an induced depth-first walk of $I(G)$, generating the spanning tree $I(D)$, which is produced by walking the nodes of $I(D)$ in the order determined by the pre-ordering of their headers in G.

Figure 3.6 shows the derived sequence of the flow graph of Figure 2.1 together with the induced depth-first walks on each derived graph.

With the pre and post orderings of $I(G)$ determined by this walk, we have (letting $\langle H, h\rangle$ and $\langle K, k\rangle$ be any two nodes in $I(G)$, and referring to the processing in Algorithm B on page 23),

$$
\begin{aligned}
\mathrm{pre}[H] < \mathrm{pre}[K] \quad &\Longleftrightarrow \quad H \text{ is first reached before } K \\
&\Longleftrightarrow \quad h \text{ is first reached before } k \\
&\Longleftrightarrow \quad \mathrm{pre}[h] < \mathrm{pre}[k]
\end{aligned}
$$

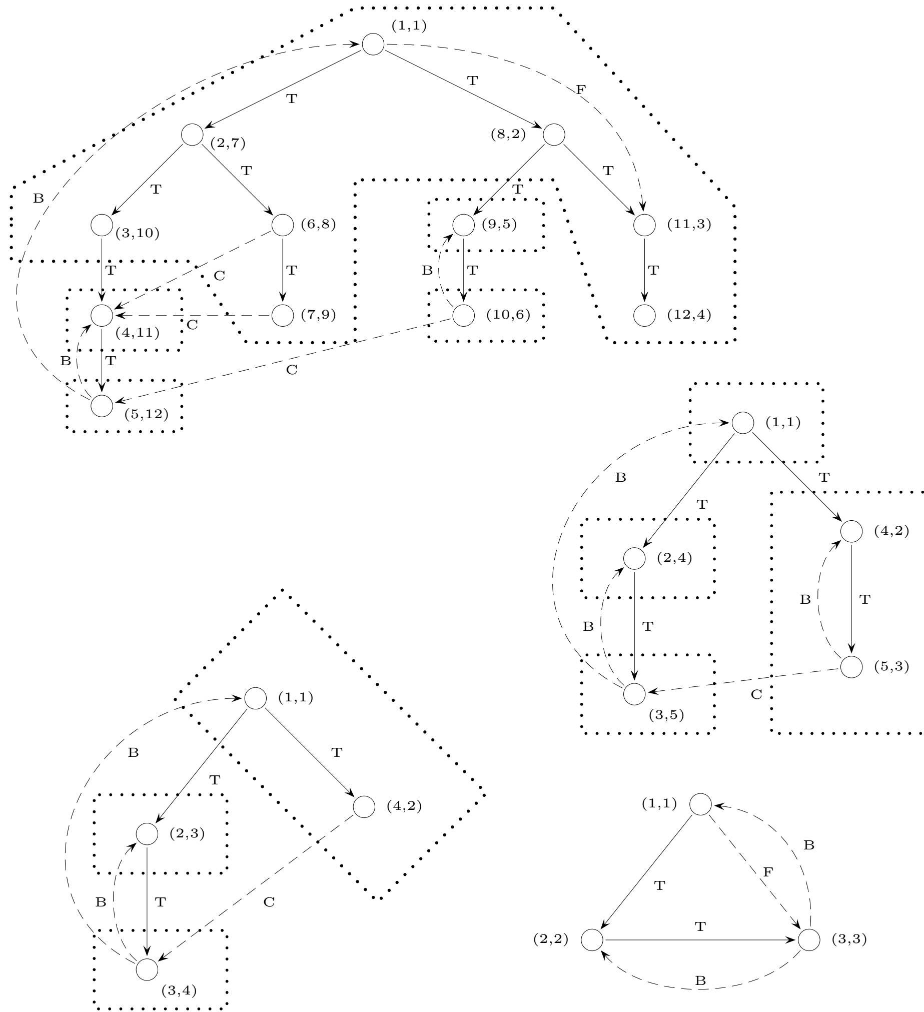
$$
\begin{aligned}
\mathrm{rpost}[H] < \mathrm{rpost}[K] \quad &\Longleftrightarrow \quad \text{all children of } K \text{ are finished being processed} \\
&\qquad\quad \text{before all children of } H \text{ are} \\
&\Longleftrightarrow \quad \text{all nodes reachable from } k \text{ are finished being} \\
&\qquad\quad \text{processed before all nodes reachable from } h \text{ are} \\
&\Longleftrightarrow \quad \mathrm{rpost}[h] < \mathrm{rpost}[k]
\end{aligned}
$$

3.23 Lemma *If $\langle H, h\rangle$ and $\langle K, k\rangle$ are disjoint intervals in G, the following are equivalent:*

1. *There is a back arc in G from H to K.*
2. *There is at least one arc in G from H to K, and all such arcs are back arcs.*
3. *There is a back arc $H \to K$ in $I(G)$.*

Proof. If there is an arc in G from H to K, then its target must be k, so it is of the form $x \to k$, where $x \in H$.

If $x \to k$ is not a back arc then $\mathrm{rpost}[x] < \mathrm{rpost}[k]$. Since in any case $\mathrm{rpost}[h] < \mathrm{rpost}[x]$ (since h dominates x), we have $\mathrm{rpost}[h] < \mathrm{rpost}[k]$, and hence $\mathrm{rpost}[H] < \mathrm{rpost}[K]$, so $H \to K$ is not a back arc.



Each node is labeled with an ordered pair of numbers (x, y), where x is the pre-order number of the node and y is the reverse post-order number of the node. Each arc is labeled T (tree arc), F (forward arc), C (cross arc), or B (back arc). Tree arcs are drawn as solid vectors; all other arcs are drawn as dashed vectors. Intervals are indicated by dotted boxes. The flow graph is not reducible.

Figure 3.6: The derived sequence for the flow graph of Figure 2.1.

Conversely, if $x \to k$ is a back arc, then k is an ancestor of x, and so there is a path in D from k to x which must therefore pass through h. That is, k is an ancestor of h. Hence rpost$[h] >$ rpost$[k]$, so rpost$[H] >$ rpost$[K]$, and so $H \to K$ must be a back arc in $I(G)$. □

3.24 Corollary *Back arcs in G fall into two disjoint classes:*

1. *Arcs within intervals in G whose target is the head of the interval.*
2. *Arcs whose image in $I(G)$ is a back arc in $I(G)$.*

3.7 Characterizations of Reducibility

In this section we will give some further characterizations of reducible flow graphs. They are all due to Hecht and Ullman ([6] and [8]; also contained in [5]).

3.25 Theorem *If $\langle G, s\rangle$ is a flow graph, the following are equivalent:*

1. *$\langle G, s\rangle$ is reducible.*
2. *The back arcs of $\langle G, s\rangle$ are unique; that is, all depth-first walks yield the same set of back arcs.*
3. *An arc $x \to y$ is a back arc $\iff y \underline{\gg} x$.*

Proof. First, if $y \underline{\gg} x$ and $x \to y$ is an arc, then since y is an ancestor of x in any tree, the arc must be a back arc in any case. Thus condition 3 really is just

3' : $x \to y$ is a back arc $\implies y \underline{\gg} x$.

If $\langle G, s\rangle$ is any flow graph, then its back arcs (under any depth-first walk) fall into two classes, by the previous corollary:

1. All arcs within intervals in G whose target is the head of the interval. These arcs have no image in $I(G)$.
2. Arcs whose image in $I(G)$ is a back arc in $I(G)$.

The arcs in class 1 are uniquely determined by the interval structure of $\langle G, s\rangle$—they will be the same for any depth-first walk of G. In addition, if $x \to y$ is a class 1 arc, then we must have $y \underline{\gg} x$.

Now we iterate this process. Precisely, let

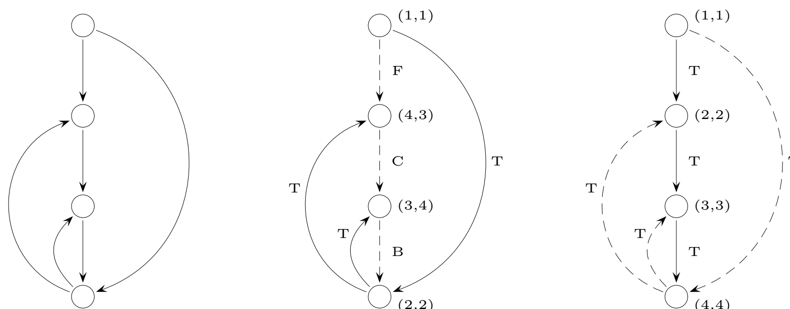
$$G = G_0, G_1, \ldots, G_n = \hat{I}(G)$$

be the derived sequence of G. At step j of the iteration ($j \geq 1$), we will consider the family of back arcs in G_j which are images of class 2 arcs $x \to y$ in G_{j-1}. Let $\mathcal{F}_j$ denote the set of arcs in this family which are class 1 arcs in G_j. $\mathcal{F}_j$ corresponds to a family of back arcs $\mathcal{B}_j$ in G, and this correspondence is determined by the interval structure of G, not by any particular walk of G. The arcs in $\mathcal{F}_j$, and hence also the arcs in $\mathcal{B}_j$ are uniquely determined by the interval structure of G_j. Similarly, for each arc $x_j \to y_j$ in $\mathcal{F}_j$ we have $y_j \underline{\gg} x_j$. Hence by Theorem 3.14, for the corresponding arc $x \to y$ in $\mathcal{B}_j$ (x and y being in G), we have $y \underline{\gg} x$.

If G is reducible, then $\hat{I}(G)$ is the trivial flow graph, which has no back arcs. Hence every back arc in G will be in one of the sets $\mathcal{B}_j$. Thus $1 \Longrightarrow 2$ and 3.

Conversely, if $\langle G, s\rangle$ is not reducible, then it contains a $(*)$ sub flow graph. If we begin a depth-first spanning tree by following the paths $SRPQ$, then all the arcs which make up path P will be tree arcs, and the last arc in Q will be a back arc. On the other hand, if we begin the tree by following the paths $STQP$, then all the arcs which make up path Q will be tree arcs and the last arc in P wil be a back arc. Thus, the back arcs will not be unique. Similarly, it is clear that property 3 does not hold for either of the back arcs in P or Q. So properties 2 and 3 each imply property 1. ☐

If $\langle G, s\rangle$ is not reducible, then not only is the set of back arcs not unique, the number of back arcs may not be unique. For instance, Figure 3.7 shows an irreducible flow graph together with two walks which yield different numbers of back arcs.



This irreducible flow graph has either 1 or 2 back arcs, depending on which way it is walked. As before, the pairs of numbers at each node are the pre-order and reverse post-order numberings of that node.

Figure 3.7: Two Walks of an Irreducible Flow Graph

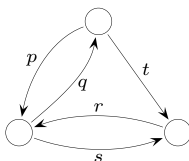
It is also possible for an irreducible flow graph (i.e. a non-trivial limit flow graph) to have a back arc $x \to y$ where y dominates x; it just can't have that property for *all* its back arcs. See Figure 3.8 for an example.

We say that a DAG of a flow graph $\langle G, s\rangle$ is a maximal acyclic sub flow graph.

3.26 Lemma *Any DAG $\langle H, E, s\rangle$ of a flow graph $\langle G, s\rangle$ includes all the nodes of G.*

Proof. If not, there must be an arc $x \to y$ with $x \in H$ and $y \notin H$. But then adjoining y to H and $x \to y$ to E still keeps $\langle H, E, s\rangle$ acyclic. ☐

3.27 Theorem *A flow graph $\langle G, s\rangle$ is reducible iff it has a unique DAG.*



An irreducible flow graph. q must be a back arc, and the target of q dominates the source of q. However (depending on which depth-first walk is used), either r or s is also a back arc, and neither of the nodes of those arcs dominates the other.

Figure 3.8: Back arcs in an Irreducible Flow Graph

PROOF. If $\langle H, E, s\rangle$ is a DAG of G, the arcs in E are tree arcs, forward arcs, or cross arcs with respect to any depth-first walk of $\langle H, E, s\rangle$. Any arc $x \to y$ of G which is not in E must be a back arc with respect to such a walk, because otherwise it could be added to E and leave $\langle H, E, s\rangle$ acyclic. Thus, E consists of all the arcs in G except the back arcs. Hence $\langle H, E, s\rangle$ is unique iff the set E is unique iff the set of back arcs of G is unique iff G is reducible. □

3.28 Theorem *A flow graph $\langle G, s\rangle$ is reducible iff its arcs can be partitioned into two sets A_1 and A_2 such that $\langle G, A_1, s\rangle$ is a DAG of G and for each arc $x \to y$ in A_2, y dominates x in G.*

PROOF. If G is reducible, just let A_2 be the (unique) set of back arcs of G.

Conversely, if G is not reducible, it has at least two DAGS, D_1 and D_2, say. If $x \to y$ is an edge which is in D_1 but not in D_2 (there must be at least one such edge), then y could not dominate x, for if it did, D_1 would not be acyclic. □

3.29 Theorem *A flow graph $\langle G, s\rangle$ is reducible iff its arcs can be partitioned into two sets A_1 and A_2 such that $\langle G, A_1, s\rangle$ is a DAG D of G and for each arc $x \to y$ in A_2, y dominates x in D.*

Remark The only difference between this theorem and the last is that we insist that y dominate x in D, rather than in G. Since there are fewer edges in D than in G, this condition is stronger in one direction, and weaker in another.

PROOF. If G is reducible, we can get D as in the previous theorem. If $x \to y$ is in A_2, then y dominates x in G, and hence in D.

To prove the converse, we need to show that if we have a partition of the edges of G into A_1 and A_2 where the A_1 edges form a DAG D, and if y dominates x in D for all the arcs $x \to y$ in A_2 , then y dominates x in G for all those arcs. We do this now:

Let the arcs in A_2 be adjoined successively in some order to D to form the subgraphs $D = H_0, H_1, \ldots, H_n = G$ of G. (Of course all these subgraphs contain the same vertices. It is only the edges that are different.)

Let j be the first number for which there is an arc $x \to y$ in A_2 for which y does not dominate x in H_j (we know $j > 0$ if it exists at all), and let H_j be formed from H_{j-1} by adjoining the arc $u \to v$, say. Then there

is a simple path P from s to x in H_j which avoids y, and this path must include the arc $u \to v$ exactly once. Hence the initial part P_1 of this path, from s to u is composed only of arcs in H_{j-1}. Since P is simple, P_1 cannot include v. But this means that v does not dominate u in H_{j-1}, a contradiction. □

3.8 The Loop-Connectedness Number

If G is reducible, the length $dsl(G)$ of the derived sequence of G (the *derived sequence length*) can be thought of as the maximum loop nesting depth in G. A famous empirical study performed by Knuth[14] showed that $dsl(G)$ was almost always bounded by 3. For the analysis of a data-structure algorithm below (see page 81), we will find it useful to estimate a related quantity:

For any flow graph G, we define $lc(G)$ to be the largest number of back arcs (with respect to a fixed depth-first walk of G) that are contained in any cycle-free path of G. Since there are only finitely many cycle-free arcs in G, $lc(G)$ is well-defined and finite. $lc(G)$ is called the *loop-connectedness number* of G. Of course if G is reducible, $lc(G)$ is a property of G alone, and is independent of the particular depth-first walk.

As usual, $I(G)$ will denote the derived graph of G.

3.30 Lemma $\mathrm{lc}(I(G)) \geq \mathrm{lc}(G) - 1$.

Proof. Let P be a cycle-free path in G with $m = lc(G)$ back arcs. Let the back arcs in P be (in order)

$$x_1 \to r_1, x_2 \to r_2, \ldots, x_m \to r_m$$

where r_i occurs before x_{i+1} in P. Since P is simple, the $\{r_j\}$ are all distinct. By Corollary 3.21 (page 33), each r_i is therefore the header of a distinct interval.

Let P' be the sub-path of P from r_1 to r_m. The image of P' in $I(G)$ must be simple, because otherwise P' (which starts at an interval header) would have to enter the same interval twice, and so would not be simple.

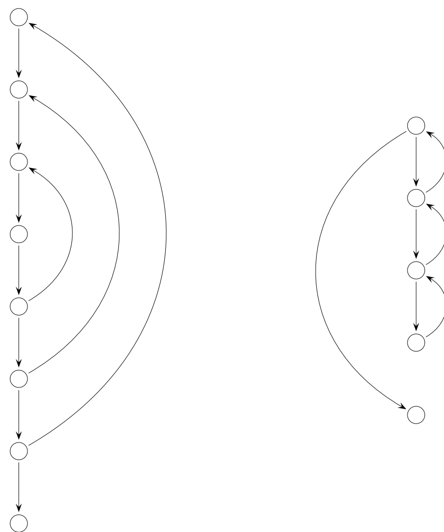
Since P is simple, and since for all i the sub-path from r_i to r_{i+1} starts in $I(r_i)$ and ends in $I(r_{i+1})$, it must enter $I(r_{i+1})$ at r_{i+1}. Thus for $1 \leq i < m$, x_{i+1} is not in $I(r_{i+1})$, and so the image of $x_{i+1} \to r_{i+1}$ in $I(G)$ is a back arc. Thus the image of P' in $I(G)$ contains at least $m-1$ back arcs. □

3.31 Theorem *If G is a reducible flow graph with loop-connectedness number* $\mathrm{lc}(G)$ *and derived sequence length* $\mathrm{dsl}(G)$*, then*

$$\mathrm{lc}(G) \leq \mathrm{dsl}(G).$$

Proof. Apply the lemma repeatedly. Since G is reducible, after $dsl(G)$ iterations, we will arrive at a trivial flow graph for which both dsl and lc are 0. But we subtracted 1 at most $dsl(G)$ times from $lc(G)$ to get to 0. □

In fact, $lc(G)$ can be much less than $dsl(G)$. For instance, Figure 3.9 shows two flow graphs. The graph on the left corresponds to three nested *repeat* loops (in the sense of Pascal; the loop always iterates at least once, and the test is at the bottom of the loop); the graph on the right corresponds to three nested *while* loops (equivalently, Fortran *do* loops). In both cases, $dsl(G) = 3$. But for the *repeat* loop nest, $lc(G) = 1$, while for the *while* loop nest, $lc(G) = 3$. Thus, *while* loops keep adding to $lc(G)$, but nests of *repeat* loops really have no effect on it.



The flow graph on the left represents three nested *repeat* loops (with the test at the bottom of the loop); it has $dsl(G) = 3$ and $lc(G) = 1$. The flow graph on the right represents three nested *while* loops (equivalently, Fortran *do* loops); it also has $dsl(G) = 3$ but has $lc(G) = 3$.

Figure 3.9: Nested Loops

3.9 Interval Nesting

Of course, there is no such thing as interval nesting. We know that intervals form a disjoint partition of G. However, each interval in the derived graph $I(G)$ is composed of a number of intervals in G, and so on; so in this sense there is a nesting of lower order intervals in higher order ones. This use of the term "nesting" reflects what we mean when we talk about loop nesting. See, for instance the flow graph on the left of Figure 3.9.

Let

$$G = G_0, G_1, \ldots, G_n = \hat{I}(G)$$

be the derived sequence of G. Let us refer to an interval in G_j as an interval of order $j+1$. (It is an element of G_{j+1}.) Each such interval can be considered as being composed of a certain number of elements of G, and in a similar way, the header of such an interval can be associated with an element of G (which is the header of an ordinary, or order-1, interval of G). Naming intervals by their headers and their order, we can indicate the nesting of lower order intervals in higher order intervals by means of a tree.

For instance, Figure 3.10 shows the derived sequence of a reducible flow graph. Figure 3.11 shows the corresponding nesting tree for intervals of all orders.

Now let us identify intervals of different orders which share the same header. We will do this by removing

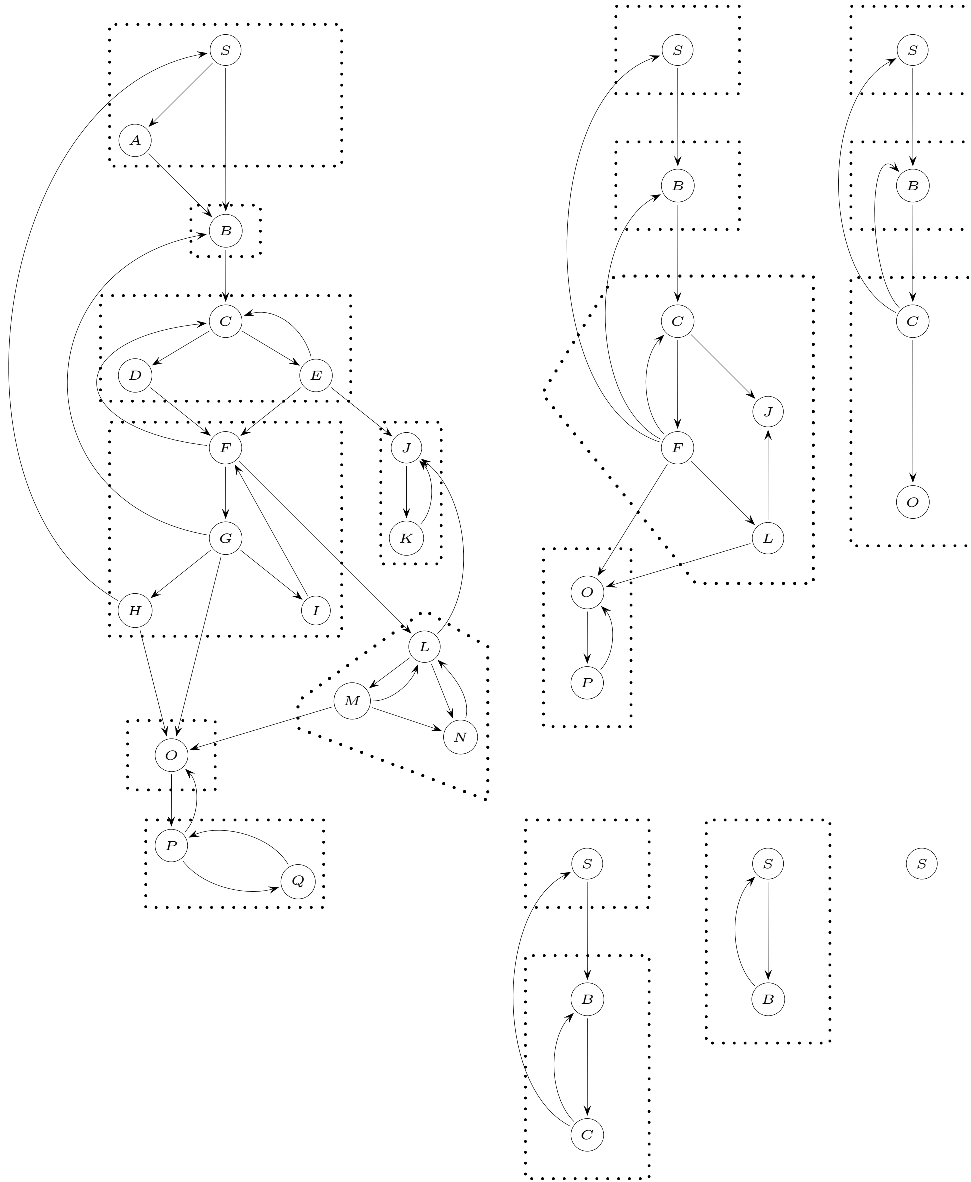
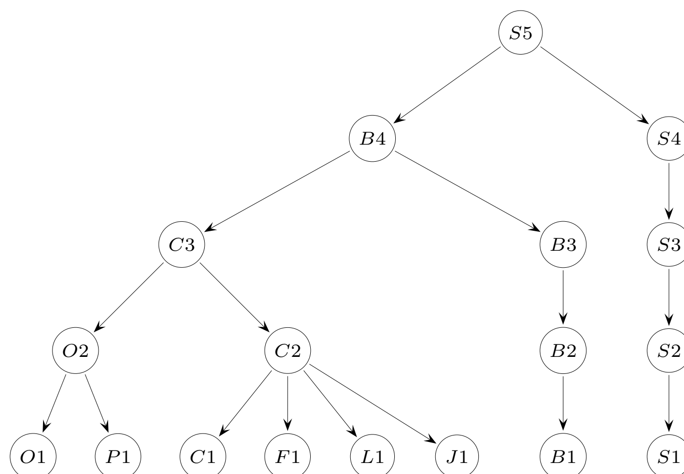


Figure 3.10: Reduction of a Flow Graph



Each interval is named by the node in G which corresponds to its header, followed by the number which is the order of the interval.

Figure 3.11: Tree Showing the Nesting of Intervals of all Orders for the Reducible Flow Graph of Figure 3.10

from the tree in Figure 3.11 all but the highest order interval for each header. The tree that remains is called the *interval nesting tree*, and is illustrated on the right of Figure 3.12. We can think of each interval in this tree as consuming all its children.

So far as the interval nesting tree is concerned, we identify each header node in G with a corresponding interval (which may be a higher order interval—for instance the interval corresponding to s is the higher order interval which corresponds to the entire flow graph). Note that

1. If x is an ancestor of y in the interval nesting tree, then x must be an ancestor of y in the dominator tree (that is, x must dominate y).
2. The converse is not true. For instance, F dominates L, but F is not an ancestor of L in the interval nesting tree.

Let us define a map *hdr* on G such that *hdr*(x) is the header of the interval containing x. Then in contrast to the above observation, we have:

3.32 Theorem *If G is reducible and $x \to y$ is an arc in G with $\mathrm{hdr}(x) \neq \mathrm{hdr}(y)$ (so in particular, y must be an interval header), then the following are equivalent:*

1. $y \underline{\gg} x$.

2. y is an ancestor of $\mathrm{hdr}(x)$ in the interval tree.

Proof. First of all, if $z \to w$ is any arc in G, let $z_i \to w_i$ denote its image in G_i. Of course, z_i might equal w_i, and since G is reducible, we know that in any case $z_n = w_n$.

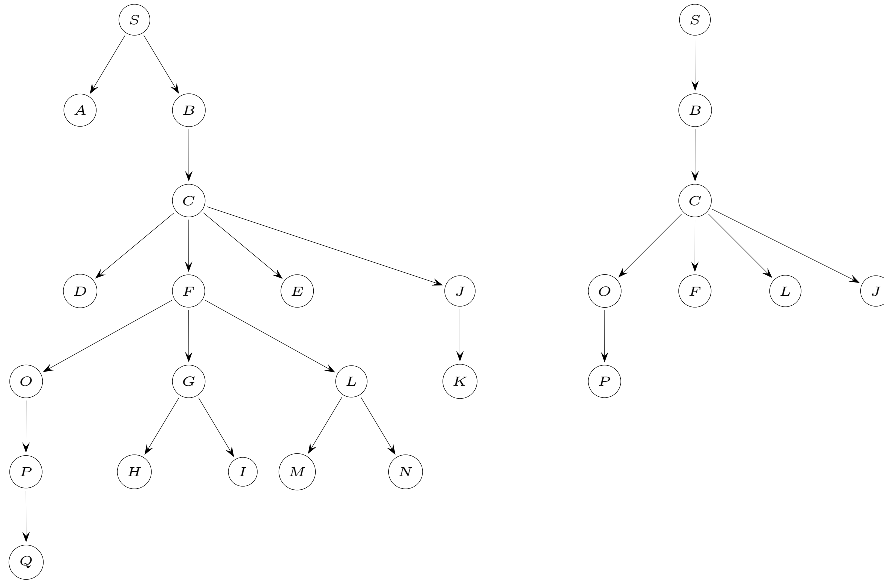


Figure 3.12: The Dominator Tree and the Interval Nesting Tree for the Flow Graph of Figure 3.10

$1 \implies 2$: We know that $x \to y$ is a back arc. As in the proof of Theorem 3.25 (page 36), the back arcs of G are partitioned into disjoint sets $\{\mathcal{B}_j\}$, where each arc $z \to w$ in $\mathcal{B}_j$ has an image $z_j \to w_j$ in G_j such that w_j is an interval header in G_j and z_j is in that interval. (That is, $z_j \to w_j$ is a back arc in an interval of G_j.) Say $x \to y$ falls in the set $\mathcal{B}_k$. Then y is the header of an interval I_y of order $k+1$.

x is a member of the set x_k, which is an element of I_y. Hence $hdr(x)$ must also be a member of x_k, and so is a descendant of x_k in the interval nesting tree. But we know that x_k is a member of I_y, and so is a descendant of y in the interval nesting tree.

$2 \implies 1$: Since y is an ancestor of $hdr(x)$ in the interval nesting tree, y dominates $hdr(x)$, and hence also dominates x. (This is just the affirmative part of the observation made earlier.) □

The arcs described by this theorem can be characterized as the arcs which are exits from intervals to larger containing intervals ("interval" in this sense again meaning interval of any order).

Figure 3.13 shows the actual interval nesting in the flow graph corresponding to the interval nesting tree of Figure 3.12 for the flow graph of Figure 3.10.

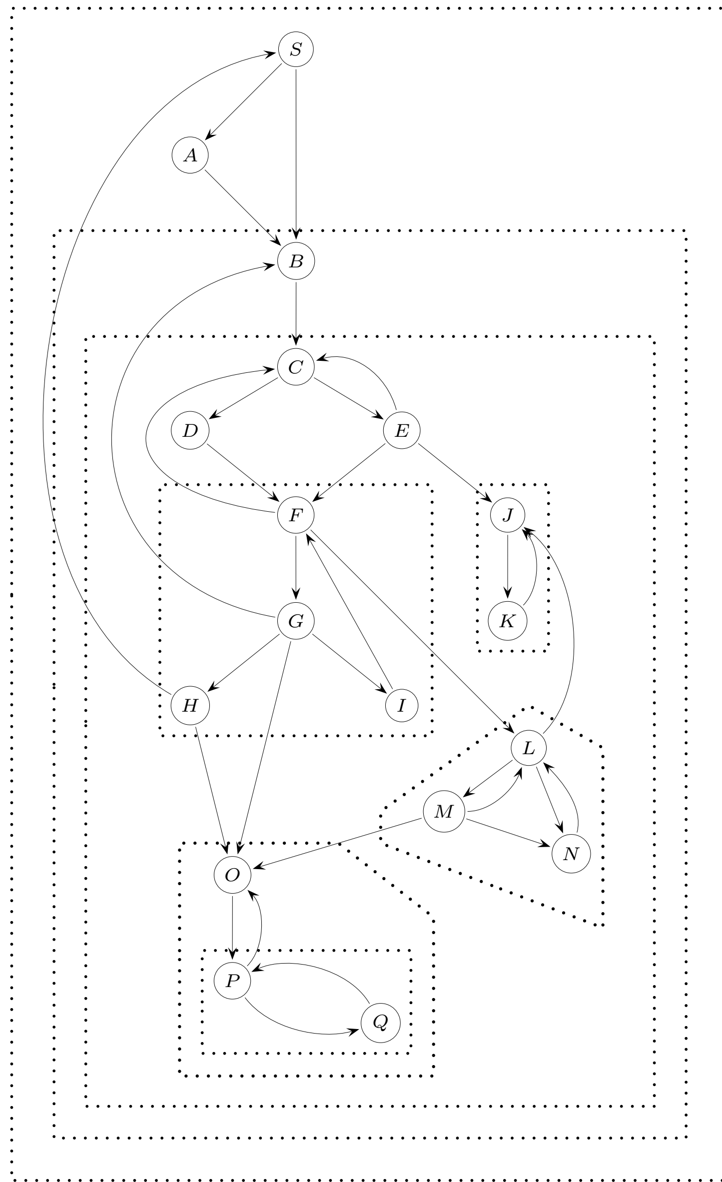


Figure 3.13: Interval Nesting for the Flow Graph of Figure 3.10

Chapter 4

Testing for Reducibility

4.1 Finite Church-Rosser Transformations

If S is a set and $\implies$ is a relation on that set (written $a \implies b$), then we say that a sequence

$$a_0 \implies a_1 \implies \ldots \implies a_k$$

is a *chain* from a_0 to a_k of length k. (We think of these chains as chains of implications or of derivations.) We write $a \overset{*}{\implies} b$ to indicate that there is a chain from a to b.

We say that the relation $\implies$ is *finite* iff for each $a \in S$ there is a bound on the length of chains $a \implies b \implies \ldots$ beginning with a. That is, $\implies$ is finite iff for each $a \in S$, there is an $n \in \mathbf{N}$ such that if

$$a \implies a_1 \implies \ldots \implies a_k$$

with all a_i distinct, then $k \leq n$. We will call the maximum length of such chains the *height* of a, and denote it by $h(a)$. It is clear that if $a \implies b$ then $h(a) \geq h(b) + 1$. If $h(x) = 0$, we will refer to x as a*limit point*, because there is no other point reachable from x via the relation $\implies$.

The completion $\widehat{\implies}$ of the relation $\implies$ is the relation defined by right-maximal chains; that is, $a \widehat{\implies} b$ iff there is a chain from a to b and there is no $c \neq b$ with $b \implies c$. Equivalently, $a \widehat{\implies} b$ iff b is a limit point and there is a chain from a to b.

We say that the relation $\implies$ is a *finite Church-Rosser* transformation iff it is finite and $\widehat{\implies}$ is a function. That is, $\implies$ is FCR iff it is finite and if $a \widehat{\implies} b$ and $a \widehat{\implies} c$ implies $b = c$.

4.1 Theorem (Aho, Sethi, and Ullman[2]) *A relation $\implies$ on a set S is a finite Church-Rosser transformation iff*

1. *$\implies$ is finite; and*
2. *for all $a \in S$, if $a \implies b$ and $a \implies c$, then there is an $x \in S$ such that $b \overset{*}{\implies} x$ and $c \overset{*}{\implies} x$.*

PROOF. A finite Church-Rosser transformation certainly satisfies conditions 1 and 2. So we only have to prove that a relation satisfying 1 and 2 is a finite Church-Rosser transformation.

We will prove this by induction on the height of a. That is, given 1 and 2, we will prove the following statement:

$$(P) \qquad \text{If } h(a) \le k, \text{ and if } a \overset{\frown}{\Longrightarrow} b \text{ and } a \overset{\frown}{\Longrightarrow} c \text{ then } b = c.$$

for all k.

(P) is trivially true for $k = 1$. Suppose it has been proved true for all $k < n$. Let a be any element of S with height n, and say $a \overset{\frown}{\Longrightarrow} b$ and $a \overset{\frown}{\Longrightarrow} c$. Thus we have chains

$$a \Longrightarrow b_1 \Longrightarrow b_2 \Longrightarrow \ldots \Longrightarrow b$$

$$a \Longrightarrow c_1 \Longrightarrow c_2 \Longrightarrow \ldots \Longrightarrow c$$

each of length $\le n$, where $h(b) = h(c) = 0$. By property 2 (since $a \Longrightarrow b_1$ and $a \Longrightarrow c_1$), we then have chains

$$b_1 \Longrightarrow \beta_2 \Longrightarrow \beta_3 \Longrightarrow \ldots \Longrightarrow x$$

$$c_1 \Longrightarrow \gamma_2 \Longrightarrow \gamma_3 \Longrightarrow \ldots \Longrightarrow x$$

We may assume without loss of generality that x is a limit point: otherwise, we could simply continue both chains in the identical manner until a limit point was attained. But now we have

$$b_1 \overset{\frown}{\Longrightarrow} b \qquad \text{and} \qquad b_1 \overset{\frown}{\Longrightarrow} x$$

and $h(b_1) < h(a) = n$, so by the inductive assumption, $b = x$. Similarly, $c = x$, and so $b = c$, and we are done. □

4.2 The Transformations $T1$ and $T2$

We define two transformations $T1$ and $T2$ on flow graphs as follows:

1. $T1$ is the removal of a self-loop. That is, if $x \to x$ is an arc in $\langle G, s\rangle$, then $T1\langle G, s\rangle$ is the flow graph whose nodes are the nodes of G and whose arcs are all the arcs of $\langle G, s\rangle$ except for the arc $x \to x$.

2. If $x \ne y$ and x is the only predecessor of y in $\langle G, s\rangle$, and if $y \ne s$, then the transformation $T2$ creates a new flow graph G' by "collapsing" y into x. Precisely, the nodes of G' are all the nodes of G except for y, and the arcs of G' are

   - all the arcs of G not involving y;
   - an arc $x \to b$ for each arc $y \to b$ in G. (In particular, if there is an arc $y \to x$ in G, then G' contains a self-loop $x \to x$.)

When $T2$ is applied in this fashion, we say that x *consumes* y.

In general, we will say that the flow graph G' is produced by *collapsing* the flow graph G if G' is produced from G by a series of transformations of the form $T1$ and/or $T2$.

Of course the transformations $T1$ and $T2$ are really not uniquely defined: if there are several self-loops in $\langle G, s\rangle$, $T1$ could be applied to any of them; and similarly for $T2$. They actually define a relation on the set of flow graphs. We can imagine applying these transformations repeatedly on a flow graph $\langle G, s\rangle$ until we can go no further. We will show that the flow graph that we get by doing this is always the limit flow graph of $\langle G, s\rangle$, no matter in what order the transformations were applied. In particular, a flow graph is reducible iff it can be transformed to the trivial flow graph by successive applications of $T1$ and $T2$, in any order.

The transformations $T1$ and $T2$ are due to Hecht and Ullman[6].

4.2 Theorem *The relation $\Longrightarrow$ defined on the set of flow graphs by the transformations $T1$ and $T2$ is a finite Church-Rosser transformation.*

Proof. First, the relation is finite, since $T1$ and $T2$ each reduce the number of arcs by at least 1. So the length of a chain starting at $\langle G, s\rangle$ can be at most the number of arcs in $\langle G, s\rangle$.

Now suppose we have $G\overset{Ti}{\Longrightarrow}L$ and $G\overset{Tj}{\Longrightarrow}M$. We want to show that the following diagram can be produced:

$$\begin{array}{ccc} G & \overset{Ti}{\Longrightarrow} & L \\ \Big\Downarrow{\scriptstyle Tj} & & \Big\Downarrow{\scriptstyle *} \\ M & \overset{*}{\Longrightarrow} & H \end{array}$$

Case I: $i = j = 1$. If $T1$ was applied to node x to yield L and to node y to yield M, then

- if $x = y$, then $L = M$, and we are done.
- otherwise, $T1$ can be applied to y in L and to x in M, yielding the same graph H in each case.

Case II: $i = j = 2$. Say x consumes y to yield L and z consumes w to yield M:

$$\begin{array}{cc} x & z \\ \big\downarrow & \big\downarrow \\ y & w \end{array}$$

We know that in any case $x \neq y$ and $z \neq w$.

- If $y = w$, then x must equal z, since y has a unique predecessor in $\langle G, s\rangle$. Hence $L = M$ already. In all the remaining cases, $y \neq w$.
- If all four nodes are distinct, then x can consume y in M and z can consume w in L to yield the same graph H.
- If $x = z$, then x can consume w in L and x can consume y in M to yield the same graph H.
- If $x = w$ but $z \neq y$, then z can consume x in L and z can consume y in M to yield the same graph H.

All other possible cases are symmetric versions of these cases.

Case III: $i \neq j$. Say x consumes y to yield L (via $T2$) and $T1$ is applied to z to yield M. y cannot equal z. Hence x can still consume y in M, and $T1$ can be applied to z (which might be x) in L, in either case yielding the same graph H. □

Therefore, we see that it does not matter in which order (or to which nodes or arcs) we apply the transformations $T1$ and $T2$—if we start with a flow graph $\langle G, s\rangle$ and keep on going until we can go no farther, we will arrive at a unique graph depending only on G.

4.3 Lemma *Any pre-interval can be collapsed to its header by a series of $T2$ operations, followed by at most one $T1$ operation. Any arcs from the original pre-interval to other nodes become arcs from the header to those same nodes.*

Proof. Let $\langle H, h\rangle$ be any pre-interval in G. There must be at least one node $x \neq h$ in H such that x has only one predecessor (y, say), for if each node in H other than h had two predecessors, then one of them would not be h, and by following arcs backward through H, always choosing the predecessor which was not h, we would eventually create a loop in H not containing h, a contradiction.

So if x is such a node, then $T2$ can be applied to collapse x into y, turning H into H', say. Any arcs from x leading outside H now become arcs from $y \in H'$ to the same target. After this application of $T2$, $\langle H', h\rangle$ is still a pre-interval, for it still has the unique entry node h, and if C is a cycle in H' not including h, then C must include y (otherwise C would have been a cycle in H). Say C looks like this:

$$c_0 \to c_1 \to \ldots \to c_{k-1} \to y \to c_{k+1} \to \ldots \to c_n = c_0.$$

There could not be an arc $y \to c_{k+1}$ in H, for then again C would be a cycle in H. Hence the arc $y \to c_{k+1}$ must come from an arc $x \to c_{k+1}$ in H. But then replacing y in C by $y \to x$ yields a cycle in H not including h, a contradiction.

If we apply this process repeatedly, we will wind up with H collapsed to its header, with possibly a self-loop at the header. This self-loop can be eliminated by applying $T1$. Thus H is replaced by its header, and every arc from an element of H to a target outside H is replaced by an arc from its header to the same target. □

While the above proof was a pure existence proof (in fact, a proof by contradiction), we will see below (Theorem 4.8 on page 50) that there actually is an algorithm for generating the sequence of T2 reductions of any pre-interval

4.4 Lemma $G \overset{*}{\Longrightarrow} I(G)$.

Proof. Applying Lemma 4.3 in turn to each interval in G yields $I(G)$. □

4.5 Theorem $G \overset{\frown}{\Longrightarrow} \hat{I}(G)$.

Remark That is, the result of applying $T1$ and $T2$ repeatedly to G until neither can be applied any more is the limit graph of G; and in particular, G is reducible iff it can be reduced to the trivial flow graph by successive applications of $T1$ and $T2$.

Proof. Applying Lemma 4.4 repeatedly, we get $G \overset{*}{\Longrightarrow} \hat{I}(G)$. So all we have to prove is that $\hat{I}(G)$ is a limit point in the set of flow graphs under the transformations $T1$ and $T2$.

First, derived flow graphs by construction do not contain self-loops. Hence $T1$ cannot be applied to $\hat{I}(G)$.

Next, since $\hat{I}(G)$ is a limit flow graph, all its pre-intervals are trivial (i.e. consist of just one node). If there were an element x of $\hat{I}(G)$ which had a unique predecessor y, then $\langle\{y, x\}, y\rangle$ would be a non-trivial pre-interval in $\hat{I}(G)$, a contradiction. Hence $T2$ cannot be applied to $\hat{I}(G)$. □

4.3 Induced Walks from $T1$ and $T2$

Now let us say that G is transformed into G' by an application of $T1$ or $T2$. If D is a depth-first spanning tree of G (corresponding to a depth-first walk of G), then D has an image D' in G'.

Case I. $T1$ was applied to get G'. Since all self-loops are back arcs, D' is the same as D, and the pre- and post-orderings of the nodes of G' are the same as those of G.

Case II. $T2$ was applied to get G'. Say $T2$ was applied to the arc $a \to b$ in G. The arc $a \to b$ was the only arc to b. Hence it must be a tree arc. So D' is D without this arc (and with a and b identified). D' is still a tree because every element except s in G' has exactly one arc in D' entering it. There is an induced walk on D' which can be described as follows:

- Each node which was visited before b is visited in the same order. This includes all the nodes visited up to a, and those children of a which were visited before b.
- Then the descendants of b are visited in the same order.
- Then the subtrees rooted at the remaining children of a (after b) are visited in the same order.
- Finally, the rest of the tree is visited in the same order.

That is, the tree is walked until b would have been reached; it is skipped over, and the remaining nodes are visited in the same order. Thus, if nodes in G are denoted by $\{x, y, \ldots\}$ and nodes in G' are denoted by $\{x', y', \ldots\}$, then

$$\begin{aligned} \text{pre}[x] < \text{pre}[b] &\implies \text{pre}[x'] = \text{pre}[x] \\ \text{pre}[x] > \text{pre}[b] &\implies \text{pre}[x'] = \text{pre}[x] - 1 \end{aligned}$$

If G has n nodes, then G' has $n - 1$ nodes. If the post-order number of a node is denoted by $\text{post}[x]$, then we have

$$\text{rpost}[x] + \text{post}[x] = n; \qquad \text{rpost}[x'] + \text{post}[x'] = n - 1.$$

Now

$$\begin{aligned} \text{rpost}[x] < \text{rpost}[b] &\iff b \text{ is finished being processed before } x \\ &\iff \text{post}[b] \text{ is assigned before } \text{post}[x] \\ &\iff \text{post}[x'] = \text{post}[x] - 1 \\ &\iff \text{rpost}[x'] = \text{rpost}[x] \end{aligned}$$

and similarly,

$$\begin{aligned}
\text{rpost}[x] > \text{rpost}[b] &\iff x \text{ is finished being processed before } b \\
&\iff \text{post}[x] \text{ is assigned before } \text{post}[b] \\
&\iff \text{post}[x'] = \text{post}[x] \\
&\iff \text{rpost}[x'] = \text{rpost}[x] - 1
\end{aligned}$$

4.6 Theorem *If x and y are nodes in G whose images in G' are x' and y', then*

1. $\text{pre}[x] < \text{pre}[y] \implies \text{pre}[x'] \leq \text{pre}[y']$.
2. $\text{rpost}[x] < \text{rpost}[y] \implies \text{rpost}[x'] \leq \text{rpost}[y']$.

If $x' \neq y'$ then

1. $\text{pre}[x] < \text{pre}[y] \iff \text{pre}[x'] < \text{pre}[y']$.
2. $\text{rpost}[x] < \text{rpost}[y] \iff \text{rpost}[x'] < \text{rpost}[y']$.

PROOF. Follows immediately from the calculations above. □

4.7 Theorem *If $x \to y$ is an arc in G which corresponds to an arc $x' \to y'$ in G', then either they are both back arcs or neither is.*

PROOF. 1. If $T1$ was applied to yield G', the pre- and post-orderings of all the nodes are unchanged, and tree arcs are preserved, so the character of all the arcs (except of course the deleted one) is unchanged. So let us assume that $T2$ was applied to yield G'.

2. If x' and y' are identified in G', then a back arc $x \to y$ must become a self-loop $x' \to x'$ in G', which is automatically a back arc. Similarly, if x' and y' are identified in G', then a self-loop $x' \to x'$ in G' could only have come from a back arc in G.

3. If x' and y' are distinct in G', then

$$\begin{aligned}
x \to y \text{ is a back arc in } G &\iff \text{pre}[x] > \text{pre}[y] \text{ and } \text{rpost}[x] > \text{rpost}[y] \\
&\iff \text{pre}[x'] > \text{pre}[y'] \text{ and } \text{rpost}[x'] > \text{rpost}[y'] \\
&\iff x' \to y' \text{ is a back arc in } G'
\end{aligned}$$

□

We also note that x' is a descendant of y' in G' (with respect to D') iff x is a descendant of y in G (with respect to D).

Now as promised, we can show that a pre-interval can be collapsed by $T2$ transformations in an easily calculated order: the order in fact is just reverse post-order:

4.8 Theorem *If the flow graph $\langle G, s\rangle$ is numbered by a depth-first walk, then any pre-interval $\langle H, h\rangle$ in G, can be collapsed to its header by applying $T2$ transformations successively to each node in $H - \{h\}$ in reverse post-order, followed possibly by one $T1$ operation on h. Any arcs from the original pre-interval $\langle H, h\rangle$ to other nodes become arcs from the header h to those same nodes.*

PROOF. First, since h dominates every element of H, $\text{rpost}[h] < \text{rpost}[x]$ for all $x \in H$.

Let x be the element of $H - \{h\}$ such that $\text{rpost}[x]$ is smallest. If $y \to x$ is an arc, then $\text{rpost}[y] < \text{rpost}[x]$, since otherwise $y \to x$ would be a back arc, and the only back arcs in H have h as their target. But then since $\text{rpost}[x]$ is minimal, we must have $y = h$. Thus, the only arc whose target is x is $h \to x$. Hence $T2$ can be applied to collapse x into h.

By Theorem 4.6, the reverse post-ordering of the nodes of H is not changed by this application of $T2$. The same argument can then be iterated to visit all the nodes of $H - \{h\}$ in reverse post-order, collapsing each in turn into h.

As in Lemma 4.3, at most one application of $T1$ to h will then complete the collapse of $\langle H, h \rangle$ to the single node h. □

4.4 Kernels of Intervals

In general, an interval consists of zero or more loops (each including the interval header), and additional non-cyclic structures hanging off the loops (or off the header if there are no loops). The loops are the essential part of the flow graph: without loops, computations of data flow are very easy. So let us refer to the loop part of an interval as the "kernel" of an interval. First we show that the kernel exists unambiguously. Clearly we want the loop part of an interval to be strongly connected and to include the interval header. We use this idea to characterize the kernel.

4.9 Lemma *The family $\mathcal{H}$ of strongly connected subsets of a pre-interval $\langle H, h \rangle$ which include h has a maximal element which includes all the members of $\mathcal{H}$.*

PROOF. Since $\{h\} \in \mathcal{H}$, $\mathcal{H} \neq \emptyset$. Clearly the union of any two members of $\mathcal{H}$ is a member of $\mathcal{H}$. Since $\mathcal{H}$ is finite, the union of all the members of $\mathcal{H}$ is maximal and includes every member of $\mathcal{H}$. □

Now we define the *kernel* of a pre-interval $\langle H, h \rangle$ to be the maximal strongly connected subset of H which includes h. We will use the notation $k(H)$ or $k\langle H, h \rangle$ to refer to the kernel of the pre-interval $\langle H, h \rangle$.

4.10 Lemma *The kernel of a pre-interval $\langle H, h \rangle$ consists of all elements x of H for which there is a path in H from x to h.*

PROOF. Call this set C. Since $k(H)$ is strongly connected, $k(H) \subset C$. Conversely, if $x \in C$ then since $\langle H, h \rangle$ is a pre-interval, there is a path P in H from h to x. Every element y in this path is in C, since the path can be followed from y to x, and then there is a path from x to h. Hence every element of C can be reached from h by a path in C, and so C is strongly connected. By maximality, then, $C \subset k(H)$. □

In particular, by Theorem 3.20, $k(H)$ includes all the back arcs in H.

4.11 Lemma *The kernel of a pre-interval with header h is a pre-interval with header h.*

PROOF. We know that $h \in k(H)$. If $a \in k(H)$, $a \neq h$, and a is the target of an arc $b \to a$, then b must be in H. Since there is then a path in H from b to h, b must be in $k(H)$. Thus, the only entry to $k(H)$ is at h. There can be no cycles in $k(H)$ which do not include h since $k(H)$ is a subset of H. □

Thus the kernel of each interval can be collapsed by Lemma 4.3. After its kernel has been collapsed, each interval has no remaining back arcs (by Theorem 4.7) and so is a DAG. There still may be back arcs in the graph, however: these correspond to back arcs between intervals in the original graph. Let us denote the graph generated by collapsing the kernel of each interval of G by $J(G)$. Just as we iterated the map $G \to I(G)$ to get a limit graph $\hat{I}(G)$ when defining reducibility, we can iterate the map $G \to J(G)$ to get a limit graph $\hat{J}(G)$. Figure 4.1 shows the construction of $\hat{J}(G)$ for the flow graph of Figure 3.3.

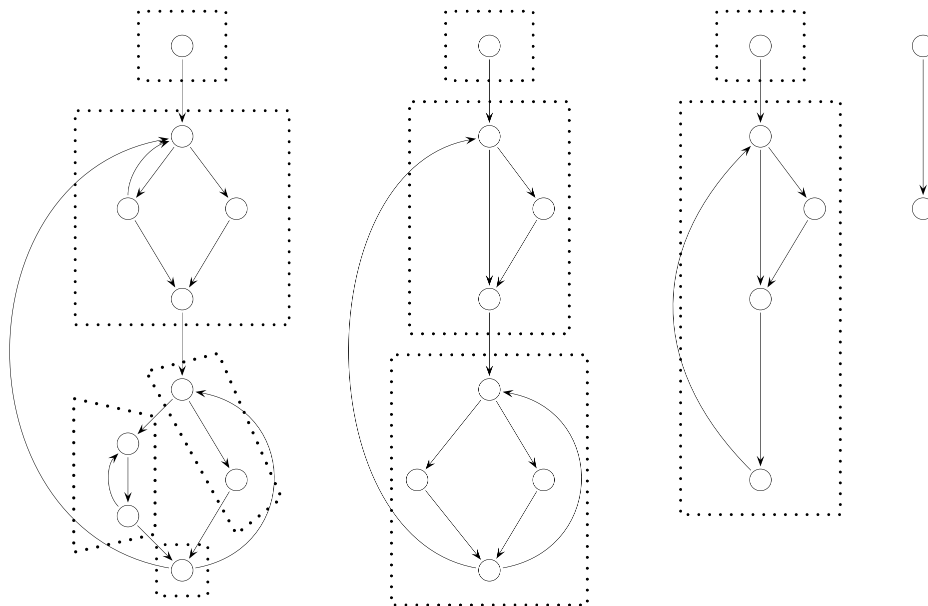


Figure 4.1: Successive Kernel Reductions for the Graph of Figure 3.3

More generally, let $T3$ be a transformation on a flow graph which consists of collapsing the kernel of 1 or more intervals of that graph. In particular, $J(G)$ is produced by the action of one version of $T3$ on G, namely the version of $T3$ which collapses the kernels of *all* the intervals of G.

It seems likely to me that the family of $T3$ transformations is a finite Church-Rosser system, and so any $T3$-limit graph of G is equal to $\hat{J}(G)$. For our purposes, however, we don't need this strong result. A simple characterization of reducibility in terms of $T3$-limit graphs is all we need:

4.12 Theorem *G is reducible iff $\hat{J}(G)$ is a DAG.*

Proof. First of all, by Theorem 4.5, G is reducible iff $\hat{J}(G)$ is reducible. So we have to show that $\hat{J}(G)$ is reducible iff it is a DAG. Since any DAG is reducible (the whole graph is an interval), we only have to show that if $\hat{J}(G)$ is reducible, it must be a DAG.

If $\hat{J}(G)$ is not a DAG, it contains a back arc (with respect to some depth-first walk). Let $x \to y$ be the back

arc such that pre[y] is maximal[1]. Let H be the union of the nodes on all simple paths from y to x. Since $\hat{J}(G)$ is assumed to be reducible, y must dominate x, and so y must dominate every node in H. Further, there cannot be any cycles in H not including y, for if so, there would be a back arc $a \to b$ with a and b in H and $b \neq y$. But since y dominates b, pre[y] < pre[b], which contradicts the maximality of pre[y] among all targets of back arcs. Hence H is a pre-interval with header y, and $x \in k(H)$. But this means that $\hat{J}(G)$ is not invariant under J, a contradiction. □

4.13 Corollary *G is reducible iff any sequence of applications of $T3$ can be extended with 0 or more applications of $T3$ to yield a DAG.*

PROOF. Since $T3$ consists of applications of $T1$ and $T2$, G is reducible iff it remains reducible after any sequence of applications of $T3$. Now apply the Theorem. □

4.14 Corollary *If $\langle G, s\rangle$ is a flow graph, the following are equivalent:*

1. *G is reducible.*
2. *Some $T3$-limit graph formed starting with G is a DAG.*
3. *Every $T3$-limit graph formed starting with G is a DAG.*

In particular, we can test a flow graph for reducibility by collapsing the kernel of just one interval, then collapsing the kernel of just one interval in the resulting graph, and so on. The graph will be reducible iff we eventually come to a DAG.

4.5 Reachunder Sets and Tarjan's Algorithm

Tarjan gave an algorithm to test for flow graph reducibility based on these ideas. His algorithm uses what we will call a "reachunder" set, which is more or less equivalent to what we have called the interval kernel—it actually could include the kernels of several intervals.

Definition *Let $\langle G, s\rangle$ be a flow graph, and let there be a depth-first spanning tree chosen for $\langle G, s\rangle$. If x is the target of a back arc in G, the* reachunder set *of x, $\rho(x)$, is the set of those elements y such that there is a simple path from y to x, the last arc of which is a back arc.*[2]

In particular, $x \notin \rho(x)$. Of course, if there is no back arc whose target is x, then $\rho(x) = \emptyset$.

Reachunder sets can be used to characterize reducibility:

4.15 Theorem *If $\langle G, s\rangle$ is a flow graph with a spanning tree D rooted at s, the following are equivalent:*

1. *G is reducible.*
2. *For all $x \in G$, x dominates every element in $\rho(x)$.*

[1] That is, maximal with respect to all back arcs. This device is due to Tarjan.

[2] The first use of the term "reachunder" appears to be in the paper by Schwartz and Sharir[16]. Tarjan doesn't give a name to the reachunder sets he uses; they occur in his paper (Tarjan[20]) as $P(x)$. The idea of the set goes back at least to Hecht[5] in his proof of our Theorem 3.18. It actually isn't necessary for that proof, as we have seen.

3. For all $x \in G$, x is an ancestor (with respect to D) of every element in $\rho(x)$.

Proof. $1 \implies 2$: Say $y \in \rho(x)$. If x does not dominate y, then there is a path in G from s to y avoiding x. If we adjoin to the end of this path the path in $\rho(x)$ from y to x, we get a path from s to x which does not contain x in its interior. Therefore, since the last arc in that path is a back arc ($z \to x$, say), we see that x does not dominate z and hence G is not reducible.

$2 \implies 3$: If x dominates an element y, then x must be an ancestor of y with respect to each spanning tree of G which is rooted at s.

$3 \implies 1$: If G is not reducible, then with respect to any given depth-first spanning tree D of G, there is a back arc $z \to x$ for which x does not dominate z. We know that $x \neq s$, since s dominates everything. Hence there is a path in G from s to z which avoids x, and hence $s \in \rho(x)$. But certainly x is not an ancestor of s. □

4.16 Lemma *If x is the node in G with the greatest pre-order number which is a target of a back arc[3], then no descendant of x is the target of a back arc.*

Proof. If b is any descendant of x, we must have $\text{pre}[x] < \text{pre}[b]$. Since x is the node with highest pre-order number which is the target of a back arc, b cannot be the target of a back arc. □

4.17 Lemma *If G is reducible and if x is the node in G with the greatest pre-order number which is the target of a back arc, then $\rho(x) \cup \{x\} = k(I(x))$.*

Proof. Say $y \to x$ is a back arc. $\rho(x) \cup \{x\}$ must have x as its only entry point because x dominates the source of every back arc (since G is reducible). Further, there can be no cycles in $\rho(x)$ because otherwise there would be a back arc in $\rho(x)$, which is impossible by the last lemma. Hence $\rho(x) \cup \{x\}$ is a pre-interval with header x, and is therefore contained in $I(x)$. Since by construction it consists of all elements z of $I(x)$ for which there is a path in $I(x)$ from z to x, Lemma 4.10 shows that $\rho(x) \cup \{x\} = k(I(x))$. □

We can now give Tarjan's algorithm for testing reducibility (Figure 4.2). The algorithm first makes a list of targets of back arcs in reverse pre-order, $v_1, \ldots, v_k$ (so that $\text{pre}[v_1]$ is maximal, and the sequence $\{\text{pre}[v_i]\}$ is decreasing). Then it computes $\rho(v_1)$. If $\rho(v_1)$ contains a non-descendant of v_1, the algorithm returns FALSE. Otherwise, $\rho(v_1)$ can be collapsed into v_1. Then $\rho(v_2)$ is computed (in the collapsed graph), and so on.

When building up ρ, each node added to ρ is checked to make sure that x is an ancestor of it. Hence by Lemma 4.16, there can be no back arcs to it. Therefore, when we are building up the set P which will eventually be $\rho(v_1)$, we only have to look at tree arcs, forward arcs, and cross arcs entering P. After $\rho(v_1)$ has been finished and collapsed into v_1, v_2 will be the node with the greatest pre-order number which is the target of a back arc (in the collapsed graph), and so the same process can be used when building up $\rho(v_2)$.

Each reachunder set is identified with its header, which is always the target of one of the original back arcs in G. As the algorithm proceeds, reachunder sets can be formed from nodes which themselves represent reachunder sets. In order to manage these set unions, the algorithm uses two functions:

FIND(x) returns the set (i.e. the node in the reduced graph) containing x.

[3] as in the proof of Theorem 4.12

```
function REDUCE(⟨G, s⟩: flow graph): boolean;
begin
    Construct a depth-first spanning tree D of G, numbering vertices from 1 to n in pre-order and reverse
        post-order and calculating ND[v] for each vertex v;
    for each vertex v (in any order) do
        Make lists of all back arcs, forward arcs, and cross arcs entering vertex v;
        Construct a set named v containing v as its only element;
        HIGHPT[v] ← 0;
    end for;
    for each vertex x in reverse pre-order do
        P ← ∅;
        for each back arc y → x do
            add FIND(y) to P;
        end for;
        Q ← P;
        -- Now construct ρ(x) by exploring backward from vertices in Q
        while Q ≠ ∅ do
            Select a vertex t ∈ Q and delete it from Q;
            for each forward arc, tree arc, or cross arc y → t do
                -- All back arcs entering t have already been collapsed
                y′ ← FIND(y);
                if pre[x] > pre[y′] or pre[x] + ND(x) ≤ pre[y′] then
                    return(FALSE);    -- G is not reducible.
                end if;
                if y′ ∈ P and y′ ≠ x then
                    Add y′ to P and to Q;
                end if;
                if HIGHPT(y′) = 0 then
                    HIGHPT(y′) ← pre[x];
                end if;
            end for;
        end while;
        -- Now P = ρ(x) in the current graph
        for s ∈ P do
            UNION(s, x, x);
        end for;
    end for;
    return(TRUE);
end
```

Figure 4.2: Tarjan's Test for Reducibility

UNION(A, B, C) creates a set C equal to the union of the sets A and B (destroying A and B in the process).

In addition, an attribute HIGHPT$[x]$ is defined for each node and computed to be the pre-order number of the first vertex (which must be a target of a back arc) into which x is collapsed (and 0 if x is not collapsed). For example, for all $y \in \rho(v_1)$, HIGHPT$[y] = \text{pre}[v_1]$.

Tarjan's algorithm gives us a convenient way to specify how a reducible flow graph can be collapsed by successive applications of the $T1$ and $T2$ transformations:

Let us define an ordering on the nodes of G, which we will call *reduction ordering*, as follows: x will precede y iff

1. HIGHPT$[x] >$ HIGHPT$[y]$; or
2. HIGHPT$[x] =$ HIGHPT$[y]$ and rpost$[x] <$ rpost$[y]$.

4.18 Theorem *The flow graph $\langle G, s\rangle$ can be reduced by a sequence of $T1$ and $T2$ transformations as follows: visit the nodes in reduction order.*

1. *Each node can be collapsed by an application of $T2$ into some other node.*
2. *All nodes with the same value of* HIGHPT *are collapsed into the same node.*
3. *When all the nodes with the same value of* HIGHPT *have been collapsed (say into node h), then a possible application of $T1$ to h can be made.*

Proof. Consider first all the nodes for which HIGHPT is greatest; this is the initial part of the sequence of nodes in reduction order. These nodes constitute the first reachunder set formed by Tarjan's algorithm. By Theorem 4.8, these nodes can be collapsed in reverse post-order (which is the same as reduction order on this set) by applications of $T2$. After they have all been collapsed into their header, an application of $T1$ to that header may be needed to delete any self-loop at the header.

We know that the reverse post-ordering of the nodes is not affected by the transformations $T1$ and $T2$ (that is, the actual numbering may change, but the ordering induced by the numbering does not). Hence this process can be iterated on each successive reachunder set until G is completely reduced. □

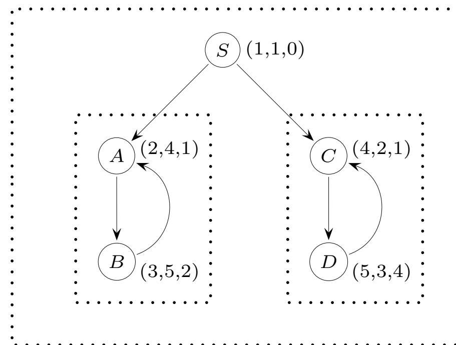
Tarjan has showed that this algorithm can be performed in time

$$O(E\alpha(E, V))$$

where V and E are the number of vertices and edges in $\langle G, s\rangle$ and where α is related to an inverse of Ackermann's function and grows so slowly that for all practical purposes it is bounded by 3.

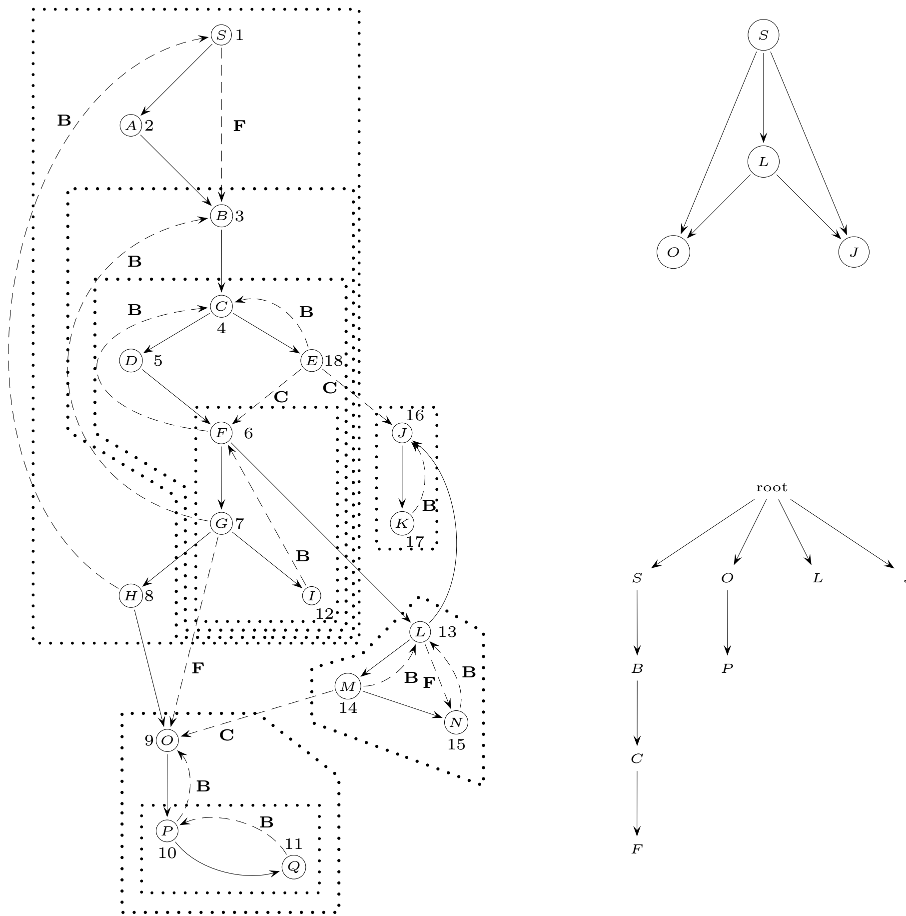
An example of the working of this algorithm is shown in Figure 4.3.

Figure 4.4 shows Tarjan's algorithm applied to the same reducible flow graph as in Figure 3.10. The limit DAG is shown in the upper right-hand corner, and the nesting tree of reachunder sets (analogous to the interval nesting tree) is shown at the lower right-hand corner. The root of the nesting tree is the limit DAG.



The ordered triples are (pre[x], rpost[x], HIGHPT[x]). The reachunder sets are outlined in dotted boxes. Reduction order is *DBCAS*.

Figure 4.3: Reduction Order Computed by Tarjan's Algorithm



The nodes are numbered in pre-order. Tree arcs are solid lines. Forward arcs (**F**), cross arcs (**C**), and back arcs (**B**) are dashed lines and are labelled. The successive reachunder sets are indicated by dotted boxes. The reachunder sets are found in the order *JLPOFCBS*.

The limit graph $\hat{J}(G)$ is shown at the upper right.

The nesting tree of reachunder sets is shown at the lower right.

Figure 4.4: Tarjan's Algorithm Applied to the Reducible Flow Graph of Figure 3.10

Chapter 5

Applications to Data-Flow Analysis

5.1 Four Data-Flow Problems

In this chapter, $\langle G, s\rangle$ will be understood to be the flow graph of basic blocks in the intermediate representation of a computer program. In addition, although we have not assumed anything about exit nodes in flow graphs (and although programs in general do not have to have unique exit nodes), we will assume that $\langle G, s\rangle$ has a unique exit node, which is reachable from every other node. This is no great restriction, for if a program has more than one exit node, we can simply create a new one and add an arc from each of the former exit nodes to the new one. And certainly we are not interested in analyzing programs which contain nodes which cannot lead to an exit. Let us denote the exit node of the program by e (for "end" or "exit"). We will denote such a flow graph by $\langle G, s, e\rangle$.

Note that a flow graph $\langle G, s, e\rangle$ is now defined perfectly symmetrically; and in fact, for each flow graph, there is a *reverse graph* $\langle G, e, s\rangle$ formed from the same nodes as G by simply reversing the direction of each arc. s and e then change roles. The reverse graph of a reducible flow graph is not necessarily reducible, however; see, for instance, Figure 5.1.

Figure 5.1: The reverse graph of a reducible flow graph may not be reducible.

For the purposes of global optimization, we wish to gather information about the use of variables in the program. The following are typical kinds of information:

Available Expressions An expression such as $x + y$ is *available* at a point P in the program iff

- the expression has been computed on each path from s to P; and

- if the expression were recomputed at P, its value would not be different from the last computed value.

For instance, not only must $x + y$ have been computed previously, but also the values of x and y must not have changed since the last computation of $x + y$ on any path to P. Any change to the values of x or y is said to "kill" the value of the expression $x + y$.

If an expression is available at a point, then in principle, it does not need to be re-evaluated at that point if it occurs there—its value may have been saved in a register. Thus, the more expressions we know are available at a point, the more useful information we have.

Reaching Definitions A *definition* of a variable is a statement which can modify the value of the variable; e.g. an assignment to the variable or a read statement. A definition of a variable v *reaches* the point P in the program iff there is a path from the definition to P on which v can be shown to be unchanged (i.e. on which there is no other definition of v; another definition of v would be said to "kill" the first definition). Given a point P in the program, and a variable v, we want to compute the set of definitions of v which reach P.

If only one definition of a variable reaches a point, then a use of that variable can be replaced by its definition—in particular, the definition may assign a constant value to the variable, and the use of the variable may thus be replaced by that constant. On the other hand, if more than one definition of a variable reaches a point, then we can't perform this kind of replacement. Thus (in contrast to the situation in the problem of Available Expressions) the fewer definitions of a variable which reach a point, the more useful information we have.

Another way that reaching definitions information can be used is this: Say E is an expression in the body of a loop. If all the reaching definitions of E are outside the loop, then E is a loop invariant and can be evaluated once before the loop is entered, rather than each time through the loop. Again, finding fewer reaching definitions constitutes finding more useful information.

Live Variables A variable is *live* at a point P of the program iff its value at P could be used along some path leaving P.

Liveness information is used in register allocation. A code generator may generate code assuming an infinite number of "virtual" registers. Each virtual register than has to be assigned to an actual machine register, and of course this has to be done in such a way that the assignments do not conflict. A modern compiler has a register allocation phase that performs this assignment.

Virtual registers can be classified as *local* (used entirely within one basic block or *global* (used in more than one basic block).

If the machine we are compiling for only has a few registers, it is common to perform only local register allocation—this means that we allocate registers within each basic block, but do not keep any values in registers between basic blocks. Thus, any global register used in a basic block will have to be written to memory at the end of the block, in order to safely preserve its value for use in some subsequent basic block. However, any virtual register that is not live at the end of a basic block does not have to be stored in memory—it can be allowed to "die" at the end of the block. So knowing that a virtual register is not live is useful information. (Note that while the problem is conventionally referred to as "live variables", the variables in question are actually the virtual registers.)

If the machine has a sufficiently large number of registers, we may also perform global register allocation. That is, we attempt to assign global virtual registers to machine registers consistently over larger segments of the program. The key point is that a virtual register has to correspond to a machine

register only at those points in the program at which the virtual register is live. Global register allocation thus is driven by liveness information. So in this case also, knowing that a virtual register is not live is useful information, because it frees up a machine register for allocation to another virtual register: if we have fewer live variables, we have more useful information.

Very Busy Expressions An expression is *very busy* at a point P of the program iff it is always subsequently used before it is killed.

If an expression is very busy at P then it can be computed at P and subsequent computations along all paths before its first use along each path can be deleted. This process is called "code hoisting", and saves space, though in general not time. (For this reason, this concept is now only of historical interest.) The more very busy expressions we have, the more code hoisting we can do, and hence the more useful information we have.

In each of these problems, we may not be able to compute all this useful information perfectly accurately. However, if we compute approximations to the useful information which yield no more than the true useful information (i.e. which are lower approximations to the true useful information), then we will at any rate not perform any unjustified program transformations. Thus, lower approximations to the true useful information in each case are safe or conservative approximations.

5.2 Data-Flow Equations

The points P of the program at which we are concerned with this information are the beginnings and the ends of basic blocks. It turns out that there are some simple relationships or constraints which we can write down. Each constraint relates four quantities:

in The information valid on entrance to a basic block.

out The information valid on exit from a basic block.

gen The information generated within the basic block.

kill The information killed or made invalid within the basic block.

In each case, the items *gen* and *kill* are quantities which we can compute in a straightforward manner for each basic block (they are functions on the set of basic blocks), and the quantities *in* and *out* (which are also functions on the set of basic blocks) are the final quantities we hope to arrive at.

For example, for the Available Expressions problem, for a given expression and a given basic block, *in* for that basic block is a boolean quantity which is TRUE iff that expression is available on entrance to that basic block.

In general, for each instance of the four examples of data-flow information given above (that is, for each expression, or each definition, etc.), these four quantities are boolean functions on basic blocks which are TRUE as follows (N.B. the precise definitions of *gen* and *kill* in each case are determined by the data-flow equations subsequently presented; in particular, the definitions for the Live Variables and Very Busy Expressions problems are tailored to reflect their use in "bottom-up" equations, as we will see below):

Available Expressions For a given expression,

in The expression is available on entrance to the basic block.

out The expression is available on exit from the basic block.

gen There is an explicit computation of the expression in the basic block which is not followed in the same basic block by a subsequent change to the value of any variable in the expression (which would change the value of the expression if it were recomputed).

kill The value of some variable in the expression changes within the basic block (so that if the expression were recomputed, its value might change), and the expression is not subsequently computed in the basic block.

Reaching Definitions For a given definition,

in The definition reaches the entrance to the basic block.

out The definition reaches the exit from the basic block.

gen The definition is produced within the basic block and not subsequently killed in the same basic block (by another definition of the same variable).

kill The definition is killed within the basic block (by another definition of the same variable within that block).

Live Variables For a given variable,

in The variable is live on entrance to the basic block.

out The variable is live on exit from the basic block.

gen The variable is used within the basic block prior to any definition of that variable within the block. Also called *use*.

kill The variable is defined within the basic block prior to any use of the variable within the basic block. Also called *def*.

Very Busy Expressions For a given expression,

in The expression is very busy on entrance to the basic block.

out The expression is very busy on exit from the basic block.

gen The expression is used within the basic block prior to any definition of any variable in that expression within the basic block. Also called *use*.

kill The value of some variable in the expression changes within the basic block (so that if the expression were recomputed, its value would be different) prior to any use of the expression in the block.

Note that in every case, $gen \wedge kill = \text{FALSE}$. This is a convention: we don't allow information to be both generated and killed within one basic block.

We now can write down some equations involving these quantities. For each basic block b, let $\text{pred}(b)$ denote the family of basic blocks which are predecessors of b (i.e. those basic blocks x for which $x \to b$ is an arc in the flow graph); and let $\text{succ}(b)$ denote the family of basic blocks which are successors of b. We have three kinds of equations:

Boundary value equations These define values at the entry or exit blocks of the program.

Propagation equations These tell how values are propagated from one basic block to another. These are also called *confluence* equations.

Transfer equations These tell how values are changed by a basic block.

There are two general types of data-flow problems: those in which information is propagated in the direction of program execution (*forward equations*), and those in which equation is propagated in the reverse direction (*backward equations*). In general, we let *in*, *out*, etc. be represented as boolean vectors (usually referred to as bit-vectors), with one dimension allocated to each expression, definition, or variable in the problem. Since boolean vectors are equivalent to subsets, we can denote the vector which is all FALSE by $\emptyset$, $\mathbf{0}$, or $\perp$. Although we will not need it till later, we will denote the vector which is all TRUE by $\mathbf{1}$ or $\top$. Similarly, the boolean operator $\vee$ will be denoted by $\cup$, and $\wedge$ will be denoted by $\cap$. We can therefore interpret the variables *in*, *out*, *kill*, and *gen* as the sets of expressions, definitions, or variables which are characterized by the conditions stated above.

Forward Equations

Available Expressions and Reaching Definitions are both examples of data-flow problems which lead to forward equations:

Available Expressions: The equations are

$$in(s) = \emptyset$$

$$in(b) = \bigcap_{x \in \mathrm{pred}(b)} out(x) \quad \text{for } b \neq s$$

$$out(b) = (in(b) - kill(b)) \cup gen(b)$$

Reaching Definitions: The equations are

$$in(s) = \emptyset$$

$$in(b) = \bigcup_{x \in \mathrm{pred}(b)} out(x) \quad \text{for } b \neq s$$

$$out(b) = (in(b) - kill(b)) \cup gen(b)$$

Thus, forward equations are characterized by passing information from the predecessors of a node to the node itself, with a boundary condition on the entry node s. The transfer equation then computes *out* in terms of *in*.

Backward Equations

Live Variables and Very Busy Expressions are both examples of data-flow problems which lead to backward equations:

Live Variables: The equations are

$$out(e) = \emptyset$$

$$out(b) = \bigcup_{x \in \mathrm{succ}(b)} in(x) \quad \text{for } b \neq e$$

$$in(b) = (out(b) - kill(b)) \cup gen(b)$$

Very Busy Expressions: The equations are

$$out(e) = \emptyset$$

$$out(b) = \bigcap_{x \in \mathrm{succ}(b)} in(x) \quad \text{for } b \neq e$$

$$in(b) = (out(b) - kill(b)) \cup gen(b)$$

Thus, backward equations are characterized by passing information from the successors of a node to the node itself, with a boundary condition on the exit node e. The transfer equation then computes *in* in terms of *out*.

Following Hecht [5], we can make a taxonomy of the four types of data-flow problems considered here so far in the form of a table:

	Set Intersection, "and", $\forall$-problems	Set Union, "or", $\exists$-problems
Top-Down Problems (flow from predecessors)	Available Expressions	Reaching Definitions
Bottom-Up Problems (flow from successors)	Very Busy Expressions	Live Variables

We note that for set intersection problems, larger sets give us more useful information, while for set union problems, smaller sets give us more useful information.

Distributive Problems

Let us examine the equations for the Available Expressions Problem more closely. If for each basic block b we write $f_b(x) = (x - kill(b)) \cup gen(b)$, then the propagation and transfer equations for this problem can be

written together as

$$out(b) = f_b\Bigl(\bigcap_{x\in \mathrm{pred}(b)} out(x)\Bigr).$$

What this says is that if we intersect the information coming in from each predecessor of b and apply f_b to that intersection, we will get the information leaving b. Actually, this may be too conservative—it might happen in principle that each predecessor x of b transmitted distinct information $out(x)$ which nevertheless got mapped by f_b to the same element. In this case, the equations we have written would yield $out(b) = f_b(\mathbf{0})$, whereas in fact $out(b)$ should really be $\cap f_b(out(x))$, which might well properly include $f_b(\mathbf{0})$.

In fact, this is not a problem in the case of Available Expressions, because the function f_b satisfies

$$f_b(x \cap y) = f_b(x) \cap f_b(y) \tag{1}$$

which means we can intersect the $out(x)$ first before applying f_b and get the same answer as if we had first applied f_b to each $out(x)$ and then performed the intersection. Each of the four data-flow problems we have considered leads to functions f_b satisfying equations such as (1)—in the case of Reaching Definitions and Live Variables, $\cap$ is replaced by $\cup$. Functions satisfying (1) are called *distributive*, and problems leading to such functions are called distributive data-flow problems.

In general, solutions of the data-flow equations don't yield correct solutions for non-distributive data-flow problems. But they do give a conservative estimate, as we will see below.

5.3 Indeterminacy of Solutions

The data-flow equations may not have unique solutions. For example, consider the flow graph in Figure 5.2. For the Available Expressions problem restricted to one expressions, the equations for this Figure are:

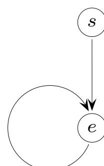


Figure 5.2: Flow Graph Illustrating Non-Unique Solution of Data-Flow Equations

$$\begin{aligned}
in(s) &= FALSE \\
in(e) &= out(s) \wedge out(e) \\
out(s) &= (in(s) \wedge \neg kill(s)) \vee gen(s) \\
out(e) &= (in(e) \wedge \neg kill(e)) \vee gen(e)
\end{aligned}$$

Let us suppose that the expression we are concerned with is evaluated in s and never evaluated again, and that it is never killed (i.e. no variable in it is every modified after the expression is evaluated. Then we must have

$$gen(s) = \text{TRUE}; \quad kill(s) = \text{FALSE}$$

$$gen(e) = \text{FALSE}; \quad kill(e) = \text{FALSE}$$

and the equations become

$$\begin{aligned} in(s) &= FALSE \\ in(e) &= out(s) \wedge out(e) \\ out(s) &= in(s) \vee \text{TRUE} = \text{TRUE} \\ out(e) &= in(e) \vee \text{FALSE} = in(e) \end{aligned}$$

After a final round of substitution, we get

$$\begin{aligned} in(s) &= FALSE \\ in(e) &= TRUE \wedge out(e) = out(e) \quad \text{(which we knew anyway)} \\ out(s) &= \text{TRUE} \\ out(e) &= in(e) \end{aligned}$$

This system of equations has two solutions: one corresponding to $out(e) = \text{FALSE}$, and the other corresponding to $out(e) = \text{TRUE}$. It is obvious, however, that $out(e) = \text{TRUE}$ is the solution we need.

The data-flow equations, therefore, should be thought of only as a set of constraints. In order to find the actual solution to the data-flow problem, we must either place further restrictions on the solutions or formulate the problem differently.

In fact, there is a different formulation of this problem which makes the solution obvious: Let us consider the set of all paths from s to e. There are an infinite number of them: $s \to e$, $s \to e \to e$, $s \to e \to e \to e$, and so forth. Each of these paths has its own data-flow properties, which can be computed by the same data flow equations, except that since each node has a unique predecessor, there is no intersection needed when passing from one node to another. It is easy to see that for each path, the expression generated in s is available at the end of the path (i.e. *out* is TRUE at the end of each path). So now here is the reformulation: The expressions available on exit from e are just those expressions which are available on exit from every path from s to e, and hence in this case, we have $out(e) = \text{TRUE}$. This is called the "meet-over-paths" solution to the data flow problem. We will look into this further in the next section.

5.4 Abstract Frameworks for Data-Flow Analysis

Before we do this, let us first observe that in some sense, these four different types of data-flow problems can be transformed into each other:

First, a backward problem can be thought of as a forward problem on the reverse flow graph. The boundary condition then becomes a boundary condition on s, rather than on e, and the roles of *in* and *out* are reversed. With these changes, the data-flow equations become identical to the forward equations.

Next, temporarily writing $\overline{in}$ for the boolean complement of *in*, and similarly for $\overline{out}$, the forward data-flow equations for set union problems (e.g. Reaching Definitions) become

$$\overline{in}(s) = \mathbf{1}$$

$$\overline{in}(b) = \bigcap_{x \in \text{pred}(b)} \overline{out}(x) \quad \text{for } x \neq s$$

$$\overline{out}(b) = (\overline{in}(b) - gen(b)) \cup kill(b)$$

which are just the forward equations for set intersection problems (e.g. Available Expressions), with the exception that the boundary condition has become an assignment of **1** instead of **0**, and the roles of *gen* and *kill* have become interchanged.

With this transformation, larger values of *in* and *out* correspond in each case to more useful information.

We can now construct an abstract model for data-flow analysis problems. As we construct the model, we will show that it includes the Available Expressions problem, and therefore the other three problems as well, since as we have seen, they are formally equivalent to it. Our abstract model will not only handle all four data-flow analysis problems presented here, but others as well which are not formally equivalent to them.

Instead of having *in* and *out* take values in the space of bit-vectors (or subsets), we will allow them to take values in a general lower semi-lattice:

A *lower semi-lattice* is a nonempty set L together with a binary operator $\wedge$ on L (called "meet"), which is commutative, associative, and idempotent. For example, the lattice of subsets of a fixed set is a lower semi-lattice with $\wedge = \cap$. If x and y are members of L, we say that $x \leq y$ iff $x \wedge y = x$.

5.1 Lemma *$\leq$ is a partial order on L.*

PROOF. Reflexivity: $x \leq x$ means $x \wedge x = x$, and this is true because $\wedge$ is idempotent.

Transitivity: If $x \leq y$ and $y \leq z$ then $x \wedge y = x$ and $y \wedge z = y$. Hence $x \wedge z = (x \wedge y) \wedge z = x \wedge (y \wedge z) = x \wedge y = x$, since $\wedge$ is associative, and so $x \leq z$; i.e. $\leq$ is transitive.

Anti-symmetry: If $x \leq y$ and $y \leq x$ then $x = x \wedge y = y \wedge x = y$, since $\wedge$ is commutative. □

In our model of data-flow analysis, L will be thought of as representing information, and to say that $x \leq y$ will be to say that x represents less information than y does.

5.2 Lemma *For all x and y in L, $x \wedge y$ is the greatest lower bound of x and y.*

PROOF. 1. Using associativity and commutativity, $x \wedge y \leq y$, and $x \wedge y \leq x$.

2. If $z \leq x$ and $z \leq y$ then $z \wedge (x \wedge y) = (z \wedge x) \wedge y = z \wedge y = z$, so $z \leq x \wedge y$. □

By induction, we can show that if $X = \{x_1, x_2, \ldots, x_n\}$ is a finite subset of L, then X has a greatest lower bound, which we denote by $\bigwedge X$, and in fact

$$\bigwedge X = x_1 \wedge x_2 \wedge \ldots \wedge x_n.$$

In practically all cases L is finite, but just in case it is not, we will go one step farther and assume that L is *complete*, by which we mean that every non-empty subset (finite or not) X of L has an inf (a greatest lower bound), which we denote by $\bigwedge X$. Completeness in general is a deep property of a lattice; it is what distinguishes the real numbers from the rationals, for example; and it is not the sort of thing you can prove by induction. All the lattices that arise in data-flow computations, however, can easily be seen to be complete. (Of course, any finite semi-lattice L is automatically complete.) If L is a complete lattice, then in particular L has a minimum element $\bigwedge L$, which we denote $\mathbf{0}$ or $\perp$. Now it turns out that much more is true: the assumption that L is complete is enough to ensure that L is actually a *lattice*, by which we mean that L also is closed under the least upper bound operation $\vee$:

5.3 Lemma *A complete lower semi-lattice is a complete lattice.*

Proof. Let A be any subset of L. Let

$$M = \{x \in L : \text{for all } a \in A, x \geq a\}.$$

Since L is complete, there is an element $m = \bigwedge M$.

Since each element a of A is a lower bound for M, it must be $\leq m$. Hence m is an upper bound for A, and by construction, it is the least upper bound for A. □

In particular, L must also have a maximum element $\mathbf{1} = \bigvee L$.[1] The maximum element is also sometimes denoted by $\top$.

We will be dealing with sets of functions having values in L. If f and g are two functions on a common domain having values in L, we say that $f \leq g$ iff $f(x) \leq g(x)$ for all x in their domain. We define the meet $f \wedge g$ of two functions pointwise: $(f \wedge g)(x) = f(x) \wedge g(x)$. A function $f : L \to L$ is *monotonic* iff $x \leq y \implies f(x) \leq f(y)$.

5.4 Lemma *$f : L \to L$ is monotonic iff for all x and y in L,*

$$f(x \wedge y) \leq f(x) \wedge f(y).$$

Proof. If f is monotonic, then since $x \wedge y \leq x$ and $x \wedge y \leq y$, we must have $f(x \wedge y) \leq f(x)$ and $f(x \wedge y) \leq f(y)$. Hence $f(x \wedge y) \leq f(x) \wedge f(y)$ by Lemma 5.2.

Conversely, if $x \leq y$, then $f(x) = f(x \wedge y) \leq f(x) \wedge f(y) \leq f(y)$, again by Lemma 5.2, so f is monotonic. □

In the data-flow problems we have been considering, L is the family of subsets of a finite set (e.g. the set of all expressions in a program). It is clearly complete, and $\perp = \emptyset$. Further, the relation $\leq$ is the same as the relation $\subseteq$.

Now for each basic block other than s in Available Expressions (our model problem) we have functions which produce *out* from *in*:

$$out(b) = f_b(in(b))$$

[1] By the proof, this amounts to saying that $\mathbf{1} = \bigwedge \emptyset$. If this seems too cute, of course, one could simply adjoin a maximum element to a semi-lattice which was otherwise complete.

where

$$f_b(x) = (x - \mathit{kill}(b)) \cup \mathit{gen}(b)$$

Let us interpret this slightly differently: think of f_b as a map from L to L which reflects the effect of the basic block b. Thus the functions $\{f_b\}$ contain all the information about each basic block and represent the information contained in the transfer equations.

In fact, we will make a further generalization: let us associate functions with arcs in the graph, rather than with nodes. For the problems we have been considering, the function $f_{x\to y}$ will be the same as what we have been denoting by f_x—that is, it will be the function reflecting the effect of the basic block x. In general, however, the functions $f_{x\to y}$ will be allowed to be different for different y. This will mean that the effect of node x may be different depending on which node y it exits to. This can be useful in two situations:

1. The test at the end of a basic block can give us information. For instance, if x exits to y if the variable $v = 4$, and otherwise exits to z, then $f_{x\to y}$ might reflect the fact that v is known to be 4, whereas $f_{x\to z}$ would not.

2. We want to allow for the possibility that the nodes in the flow graph are not themselves basic blocks but have internal structure. For instance, the flow graph might be a derived graph of another flow graph, in which case each node would actually be an interval in the other flow graph. This interval could have several exits. Associating the functions f with edges leaving nodes does not completely enable us to capture all the information due to the fact that a node could have several internal exits, but it does help a little.

To include these functions in our formal model, we will assume that we have a family $\mathcal{F}$ of functions mapping L to L such that

1. $\mathcal{F}$ contains the identity function I.

2. $\mathcal{F}$ is closed under function composition.

3. Each function in $\mathcal{F}$ is monotonic.

Such a structure $\langle L, \wedge, \mathcal{F}\rangle$ is called a *monotone data flow analysis framework*.

Let us show that Available Expressions is modeled by a monotone data flow analysis framework. The set of functions $\mathcal{F}$ will be the set of all functions from L to L of the form

$$f(x) = (x - \mathit{kill}) \cup \mathit{gen}$$

(Equivalently, $f(x) = (x \cap \mathit{kill}^c) \cup \mathit{gen}$.[2]) We get the identity function when $\mathit{kill} = \mathit{gen} = \mathbf{0}$. The fact that $\mathcal{F}$ is closed under function composition follows from the following identity, where we take $f(x) = (x \cap a) \cup b$ and $g(x) = (x \cap c) \cup d$:

$$f \circ g(x) = \Big(x \cap (a \cap c)\Big) \cup \Big((a \cap d) \cup b\Big)$$

[2] In general, if X is any set, X^c stands for the complement of that set.

Finally, every function of the form $f(x) = (x \cap a) \cup b$ is clearly monotonic, since set inclusion is equivalent to the relation $\leq$.

An *instance* of a monotone data flow analysis framework $\langle L, \wedge, \mathcal{F}\rangle$ is a flow graph $\langle G, s, e\rangle$ together with

1. an assignment $e \to f_e$ of a function $f_e \in \mathcal{F}$ to each edge e of G.
2. a value $\lambda \in L$ assigned to s. This is the boundary condition. Typically, λ is either **0** (as in Available Expressions), or **1** (as in Reaching Definitions).

5.5 Solutions of Abstract Data Flow Problems

The Meet-Over-Paths Solution

Given an instance of a monotone data flow analysis framework, we can define a function f from paths $P : s = x_0 \to x_1 \to \ldots \to x_n = b$ to L by

$$f(P) = f_{x_{n-1}\to x_n} \circ f_{x_{n-2}\to x_{n-1}} \circ \ldots \circ f_{x_0\to x_1}(\lambda).$$

Note that $f(P)$ does not include the effect of node x_n—intuitively, we are calculating data flow up through the end of x_{n-1}; that is, up to the beginning of x_n. In terms of the Available Expressions problem, we are calculating what might be denoted $in_P(b)$.

The *meet-over-paths solution* (sometimes referred to as the *mop* solution) of an instance of a data flow analysis framework is the function from G to L defined by

$$mop(b) = \begin{cases} \lambda & \text{if } b = s \\ \displaystyle\bigwedge_{\substack{\text{paths } P \\ \text{from } s \text{ to } b}} f(P) & \text{otherwise} \end{cases}$$

The number of paths may be infinite, but since L is complete, the expression on the right-hand side is well-defined.

If we knew that every legal path in the program could actually be executed, then the meet-over-paths solution would yield the true values of *in* at each basic block (and from that, we could get the values of *out*). In general, however, it is an undecidable question whether a particular path in a program will ever be executed, and so the meet-over-paths solution may not be strictly correct. If it is incorrect, however, it is only because too many paths were included, and this would make the meet-over-paths solution too small. That is, the meet-over-paths solution is a lower approximation to the real solution. As such, it contains less information than the real solution, and so is a safe, or conservative, estimate of the real solution.

The Maximal Fixed-Point Solution

Since most computer programs contain loops, the number of paths to most nodes in the flow graph is infinite, and hence in general the meet-over-paths solution is difficult or impossible to compute. We are always safe, however, if we can compute a lower approximation to it, since we will then have a lower approximation to

the real solution. It turns out that instances of monotone data-flow analysis frameworks always have such solutions, and the solutions are easily computable. These solutions can be computed in terms of the original data-flow constraint equations.

First, note that the constraint equations for our model problem can be written

$$in(s) = \lambda$$

$$in(b) = \bigcap_{x \in \mathrm{pred}(b)} (in(x) - kill(x)) \cup gen(x) \quad \text{for } b \neq s$$

where λ is a constant; it is $\mathbf{0}$ for the Available Expressions problem. Now the family $\mathcal{M}$ of functions mapping G to L is a complete lower semi-lattice under the pointwise operations. It has a maximal element; namely, the function m whose value $m(b)$ at every point b is $\mathbf{1}$. If m is a typical member of $\mathcal{M}$ and if we define a map $T : \mathcal{M} \to \mathcal{M}$ by

$$(Tm)(s) = m(s)$$

$$(Tm)(b) = \bigcap_{x \in \mathrm{pred}(b)} f_{x \to b}(m(x)) \quad \text{for } b \neq s$$

then the solutions to the model constraint equations are just those functions m which are fixed points of the transformation T under the constraint $m(s) = \lambda$. Equivalently, let $\mathcal{M}_0$ be the subspace of $\mathcal{M}$ consisting of all the functions $m \in \mathcal{M}$ such that $m(s) = \lambda$. $\mathcal{M}_0$ is invariant under T, and $\mathcal{M}_0$ is still a complete lower semi-lattice under the pointwise operations. Its maximal element is the function m defined by

$$m(b) = \begin{cases} \lambda & b = s \\ \mathbf{1} & b \neq s \end{cases}$$

The solutions to the model constraint equations are then just the fixed points of T when restricted to $\mathcal{M}_0$.

Since all the maps f_x are monotonic, the transformation T is also.

5.5 Theorem (Tarski) *If $\mathcal{L}$ is a complete lattice and T is a monotonic function on $\mathcal{L}$, then*

1. *T has a fixed point.*
2. *The set of fixed points of T has a maximal element*

Proof. Let $\mathcal{U}$ denote the set $\{x \in \mathcal{L} : T(x) \geq x\}$. Thus any fixed points of T (if there are any) will lie in $\mathcal{U}$. Further, $\mathcal{U} \neq \varnothing$ since $\mathbf{0} \in \mathcal{U}$.

Let $a = \sup \mathcal{U}$.

For each $u \in \mathcal{U}$, $a \geq u$, and so $T(a) \geq T(u) \geq u$. So we have

$$a = \sup\{u : u \in \mathcal{U}\} \leq \sup\{T(u) : u \in \mathcal{U}\} \leq T(a)$$

which shows that $a \in \mathcal{U}$.

Now we know that $a \le T(a)$, and so $T(a) \le T(T(a))$; i.e., $T(a) \in \mathcal{U}$. If in fact $a \ne T(a)$, then $a < T(a) \in \mathcal{U}$, and so a cannot be the sup of $\mathcal{U}$, a contradiction. Thus a is in fact a fixed point for T. a must be the maximal fixed point, since $\mathcal{U}$ includes all the fixed points of T. □

Thus the constraint equations have a maximal solution, which we call the *maximal fixed-point solution* and denote *mfp*. The proof we have given is non-constructive. If we restrict the semi-lattice L further, however, a constructive algorithm can be given for finding *mfp*. The algorithm is quite simple, and is really based on the essential idea of the proof—start with an element, and iterate T. In this case, however, we start with the maximal element $\mathbf{1}$ of $\mathcal{L}$.[3]

The algorithm works for any complete lower semi-lattice L with a maximal element $\mathbf{1}$ which also satisfies the *descending chain condition* (DCC): each descending sequence $x_1 > x_2 > x_3 > \ldots$ has only finitely many elements. (The number of elements does not have to be uniformly bounded over all chains, however.) Again, any finite semi-lattice (such as those characterizing the four data-flow problems we have been considering) certainly satisfies DCC; also, any semi-lattice which satisfies DCC is automatically complete.

Figure 5.3 gives an algorithm for finding the maximal fixed-point solution if L satisfies DCC.

---

begin
    $m(s) \leftarrow \lambda$;
    for each node $b \in G - \{s\}$ **do**
        $m(b) \leftarrow \mathbf{1}$;
    repeat
        for each $b \in G - \{s\}$ **do**
$$m(b) \leftarrow \bigwedge_{x \in \mathrm{pred}(b)} f_{x \to b}(m(x));$$
        end for;
    until there is no change to any $m(b)$;
end

Figure 5.3: Algorithm E:
Algorithm for Computing the Maximal Fixed-Point Solution

5.6 Theorem *If L satisfies DCC, then Algorithm E terminates and computes the maximal fixed-point solution of the data-flow constraint equations.*

Proof. Let m_0 denote the function m as initialized, and let m_j denote the value of m after the *while* loop has been executed j times. By induction, $m_{j+1}(b) \le m_j(b)$ for all b and j. Since L satisfies DCC and G is finite, the algorithm must terminate in a finite number of steps.

Let m be the final value computed by the algorithm. By construction, $Tm = m$ and $m(s) = \lambda$, so m is a fixed point and a solution to the data-flow constraint equations. If a is any other solution, then since $a \le m_0$, we have $a = Ta \le T(m_0) = m_1$, and by induction, $a \le m$. Thus, m is the maximal fixed-point solution. □

[3] One could also prove Tarski's theorem by approximating from above, but the proof becomes slightly more technical—letting $\mathcal{P}$ denote the (possibly empty) set of fixed points of T, we define $\mathcal{U}$ to be the set of those points $x \in \mathcal{L}$ such that $x \ge T(x)$ *and* $x \ge$ every element of $\mathcal{P}$. The proof then goes through much as above. The proof we have given is the one given by Tarski in his original paper [24]. Tarski actually proved somewhat more: he proved that the set $\mathcal{P}$ of fixed points of T was itself a lattice.

5.7 Corollary *The result produced by Algorithm E is independent of the order in which the nodes are processed in the second* for *loop.*

We say that a function $f : L \to L$ is *distributive* (or a *semi-lattice homomorphism*) iff for all x and y in L,

$$f(x \wedge y) = f(x) \wedge f(y).$$

By Lemma 5.4, every distributive function is monotonic. A distributive data-flow analysis framework is one in which all the functions in $\mathcal{F}$ are distributive. It is easy to check that the functions of the form $f(x) = (x \cap a) \cup b$ are distributive, so the four data-flow problems we have been considering are all distributive.

We can now derive the relationship between the *mfp* and *mop* solutions. The facts are that in general, $mfp \le mop$, and if $\langle L, \wedge, \mathcal{F}\rangle$ is distributive and satisfies DCC, $mfp = mop$.

5.8 Lemma *If $\langle L, \wedge, \mathcal{F}\rangle$ is a monotone data-flow analysis framework, and if f is any function in $\mathcal{F}$ and X is any subset of L, then*

$$f\Bigl(\bigwedge X\Bigr) \le \bigwedge_{x \in X} f(x).$$

If in fact $\langle L, \wedge, \mathcal{F}\rangle$ is a distributive data-flow analysis framework which satisfies DCC, then

$$f\Bigl(\bigwedge X\Bigr) = \bigwedge_{x \in X} f(x).$$

Proof. In any case, we have

$$\bigwedge X \le x$$

for each $x \in X$, and so since f is monotonic,

$$f\Bigl(\bigwedge X\Bigr) \le f(x)$$

for each $x \in X$, and therefore

$$f\Bigl(\bigwedge X\Bigr) \le \bigwedge_{x \in X} f(x).$$

To prove the converse, since we are now assuming that $\langle L, \wedge, \mathcal{F}\rangle$ satisfies DCC, there is a finite subset $\{x_1, x_2, \ldots, x_n\}$ of X such that

$$\bigwedge X = x_1 \wedge x_2 \wedge \ldots \wedge x_n.$$

Thus,

$$\begin{aligned} \bigwedge_{x \in X} f(x) &\le f(x_1) \wedge \ldots \wedge f(x_n) \\ &= f(x_1 \wedge \ldots \wedge x_n) \quad \text{since } f \text{ is distributive} \\ &= f\Bigl(\bigwedge X\Bigr) \end{aligned}$$

□

5.9 Theorem (Kam and Ullman[10]) *If* $\langle L, \wedge, \mathcal{F}\rangle$ *and* $\langle G, s, e\rangle$ *constitute an instance of a monotone data-flow analysis framework, then* mfp $\leq$ mop.

PROOF. We let $\pi(b)$ denote the set of all paths from s to b, and $\pi_j(b)$ denote the set of paths from s to b of length $\leq j$. Thus, $\pi(b) = \cup \pi_j(b)$, and

$$mop(b) = \bigwedge_{P\in\pi(b)} f(P) = \bigwedge_j \bigwedge_{P\in\pi_j(b)} f(P).$$

We will show by induction that

$$(*) \qquad mfp(b) \leq \bigwedge_{P\in\pi_j(b)} f(P)$$

for all j, which will prove the theorem.

For $j = 0$, the right-hand side equals $\mathbf{1}$ except at s, where it equals $\lambda = mfp(s)$.

Now suppose we know that (*) is true for $j = n$; we want to show it is true for $j = n + 1$. If $b = s$, there is again nothing to prove. Otherwise, we have

$$\begin{aligned} mfp(b) &= \bigwedge_{x\in \mathrm{pred}(b)} f_{x\to b}(mfp(x)) \\ &\leq \bigwedge_{x\in \mathrm{pred}(b)} f_{x\to b}\Big(\bigwedge_{P\in\pi_n(x)} f(P)\Big) \\ &\leq \bigwedge_{x\in \mathrm{pred}(b)} \bigwedge_{P\in\pi_n(x)} f_{x\to b}\circ f(P) \\ &= \bigwedge_{P\in\pi_{n+1}(b)} f(P) \end{aligned}$$

□

Thus for instances of monotone data-flow analysis frameworks, the maximal fixed-point solution is easily computable and is a conservative estimate for the true solution of the data-flow problem.

In fact, for the most common data-flow problems, including the ones we started with, the *mfp* and *mop* solutions are equal, as we will now show:

5.10 Theorem *In an instance of a distributive data-flow analysis framework satisfying DCC,* mfp = mop.

PROOF. In any case $mop(s) = \lambda = mfp(s)$. For $b \neq s$, and with the same notation as in the previous proof,

we have $mop(b) = \bigwedge_{P \in \pi(b)} f(P)$. Hence

$$\begin{aligned} T(mop(b)) &= \bigwedge_{x \in \text{pred}(b)} f_{x \to b}(mop(x)) \\ &= \bigwedge_{x \in \text{pred}(b)} f_{x \to b}\Big(\bigwedge_{P \in \pi(x)} f(P) \Big) \\ &= \bigwedge_{x \in \text{pred}(b)} \bigwedge_{P \in \pi(x)} f_{x \to b} \circ f(P) \\ &= \bigwedge_{P \in \pi(b)} f(P) = mop(b) \end{aligned}$$

so $mop(b)$ is a fixed point of T and so is $\leq mfp(b)$. □

This theorem was in essence first proved by Kildall in his Ph.D. thesis ([12]; summarized in [13]).

5.6 Two More Data-Flow Analysis Problems

Constant Folding

To keep the notation simple, in this section, we will revert to the convention that transfer functions are associated with nodes, rather than with edges.

A form of constant folding can be handled as a data-flow analysis problem.

Let K be the set of variables in the program we are considering. The set of possible constant values of any variable in K will be denoted by V (so typically, V is defined to be **R**, **Z**, or **C**). We will adjoin a "bottom" value $\perp$ and a "top" value $\top$ to V to create an augmented value space V^*. V^* is ordered as follows: there is no order relation between members of V, but $\perp$ is less than every member of V and $\top$ is greater than every member of V. (So in particular, any chain in V^* has length ≤ 3.)

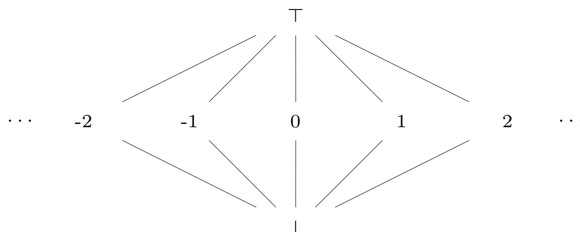


Figure 5.4: The constant folding lattice V^*.

The semi-lattice L will consist of all functions from K to V^*. L will be ordered in the obvious way pointwise (using the ordering of V^* defined above). **0** is just the function which is $\bot$ everywhere, and **1** is the function which is $\top$ everywhere. L has infinitely many elements. However, since there are finitely many variables in the program—n, say—the length of any chain of elements in L is $\leq 2n+1$; and hence L satisfies DCC, and in particular is complete.

The intuition for this is that if ϕ is a solution of our data-flow analysis problem, and if x is a variable in the program, then $\phi(x)$ has a value in V iff x is known to be constant, and the value of $\phi(x)$ is that constant. $\phi(x) = \bot$ will mean that the variable x is not constant. $\phi(x) = \top$ will mean that nothing is known about x—this happens only during the course of the algorithm; when the algorithm is finished the value $\top$ will never occur. (The idea is that the value of ϕ at each local variable will be initialized to $\top$; clearly it would be a semantic violation to use such a variable prior to assigning to it.)

Thus, every statement s in the program will have a function $f_s \in \mathcal{F}$ associated with it, and the transfer function for a basic block will be the functional composition of those functions corresponding to the statements in that block. f_s will be the identity function unless s is an assignment statement of one of the following forms (where x and y_j denote variables in the program):

$x \leftarrow a$ where a is a constant. Then f_s is defined by

$$f_s(\phi)(w) = \begin{cases} \phi(w) & \text{if } w \neq x \\ a & \text{if } w = x \end{cases}$$

$x \leftarrow \theta(y_1, \ldots, y_n)$ where the right-hand side is an expression involving only the variables $y_1, \ldots, y_n$ and possibly some constants. Then f_s is defined by

$$f_s(\phi)(w) = \begin{cases} \phi(w) & \text{if } w \neq x\text{; and otherwise,} \\ \theta(\phi(y_1), \ldots, \phi(y_n)) & \text{if all } \phi(y_j) \text{ are in } V \\ \bot & \text{if any } \phi(y_j) = \bot \end{cases}$$

We don't have to bother considering what happens when any $\phi(y_j) = \top$, since as we noted above, this would be semantically invalid.

We let $\mathcal{F}$ be the set of functions generated by the identity and these functions and closed under functional composition. To show that $\langle L, \wedge, \mathcal{F} \rangle$ is a monotone data-flow analysis framework, it will be enough to show that each of these generating functions is monotonic.

Suppose then that $\phi \leq \psi$ are members of L. First, say s is a statement of the form $x \leftarrow a$. Then we have

$$f_s(\phi)(w) = \phi(w) \leq \psi(w) = f_s(\psi)(w)$$

if $w \neq x$, and they are equal if $w = x$; hence f_s is monotonic.

Next, say s is a statement of the form $x \leftarrow \theta(y_1, \ldots, y_n)$. If $w \neq x$ then as before, $f_s(\phi)(w) \leq f_s(\psi)(w)$. Otherwise, $w = x$. Now if any of the $\phi(y_j) = \bot$, then $f_s(\phi)(w) = \bot \leq f_s(\psi)(w)$. Otherwise, none of the $\phi(y_j) = \bot$ and therefore none of the $\psi(y_j) = \bot$. As we noted before, we can assume that all $\phi(y_j)$ and $\psi(y_j)$ are in V. But then since each $\phi(y_j) \leq \psi(y_j)$, we must have $\phi(y_j) = \psi(y_j)$ for all j, so $f_s(\phi)(w) = f_s(\psi)(w)$.

Thus $\langle L, \wedge, \mathcal{F} \rangle$ is a monotone data-flow analysis framework. It is not, however, distributive, as the program in Figure 5.5 shows.

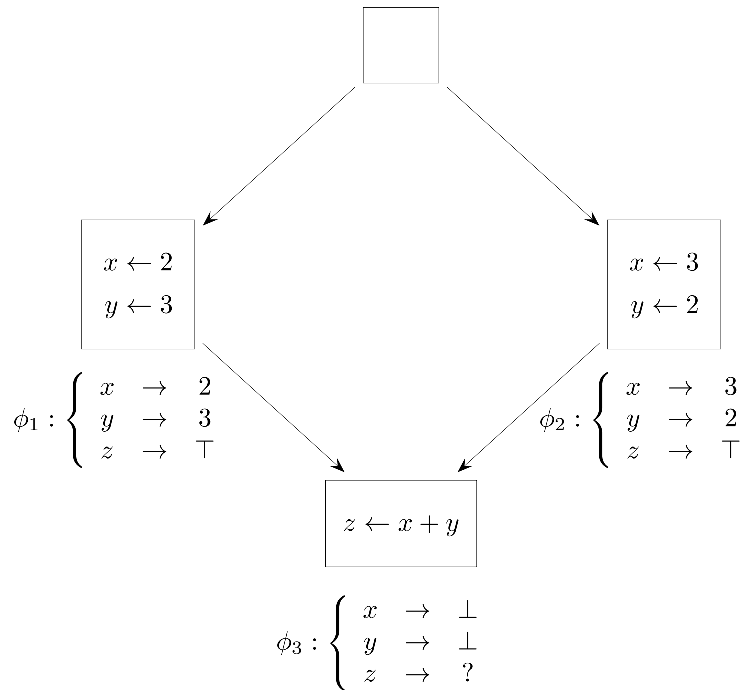


Figure 5.5: Non-Distributivity for Constant Folding

In this figure, $\phi_1 \wedge \phi_2$ is $\bot$ at both x and y and $\top$ at z. Hence, $f_{z \leftarrow x+y}(\phi_1 \wedge \phi_2)(z) = \bot$. On the other hand, $f_{z \leftarrow x+y}(\phi_1)(z) = f_{z \leftarrow x+y}(\phi_2)(z) = 5$, so $f_{z \leftarrow x+y}(\phi_1 \wedge \phi_2) \neq f_{z \leftarrow x+y}(\phi_1) \wedge f_{z \leftarrow x+y}(\phi_2)$; i.e. the framework is not distributive.

Finding Dominators

The problem of finding dominators in a flow graph can be cast into the form of a distributive data-flow analysis problem, as follows:

Given a flow graph $\langle G, s, e \rangle$, the lattice L will consist of the set of all subsets of G, ordered by inclusion. In particular, $\mathbf{0} = \emptyset$, and $\mathbf{1} = G$. The function f_n corresponding to a node $n \in G$ will be

$$f_n(S) = S \cup \{n\}.$$

where S is any subset of G.

It is easy to see that functions of this form (together with the identity function) constitute a distributive

data-flow analysis framework. It is then obvious that the meet-over-paths solution (with the boundary value condition $S_s = \{s\}$) yields precisely the set of dominators at each node, since it consists of those nodes which occur on every path from s to that node.

This gives us a reasonably good way of finding dominators in directed graphs. And although it is not the most efficient way, it is surely the simplest one conceptually.

Also, if we make the lattice meet operation $\wedge$ correspond to $\cup$ instead of $\cap$, then the same data-flow analysis problem, instead of yielding all the dominators of a node, yields all the "ancestors" of the node (with respect to the whole graph; that is, all the nodes from which there is a path to the node under consideration).

5.7 How Many Iterations are Needed?

Many data-flow analysis frameworks have a property which enables us to show that the iterative Algorithm E for finding *mfp* will converge quickly. There are various forms of this property, but they all say in one form or another that in finding the *mop* solution to a data-flow problem, we can restrict ourselves to a finite number of paths.

Let us relax our assumption that all paths start at the entry node s of G. In general, our iterative algorithms start with an initial value $\alpha(x) \in L$ at each node $x \in G$, where $\alpha(s) = \lambda$ and otherwise $\alpha(x) = \mathbf{1}$; for a path

$$P : x_0 \to x_1 \to \ldots \to x_n,$$

we define

$$f(P) = f_{x_{n-1}\to x_n} \circ f_{x_{n-2}\to x_{n-1}} \circ \ldots \circ f_{x_1\to x_0}(\alpha(x_0)).$$

Now each $P \in \pi(b)$ either starts at s or can be extended to a path from s to b: say it does not start at s and the new path is

$$P' : s = y_0 \to y_1 \to \ldots \to y_m = x_0 \to x_1 \to \ldots \to x_n = b.$$

Let us write

$$Q : s = y_0 \to y_1 \to \ldots \to y_m = x_0.$$

Then we have $f(Q) \leq \mathbf{1} = \alpha(x_0)$, and so $f(P) \geq f(P')$.

Thus, adding all these new paths (the ones not starting at s) to the meet in computing *mop* cannot change the value of *mop*, since each such new path is bounded below by a path already in the meet (i.e., already starting at s).

Now suppose it is actually true that a finite number of paths suffices to compute *mop*; in fact, say we happen to know that for each $b \in G$

$$mop(b) = \bigwedge_{P\in\pi(b)} f(P)$$

where $\pi(b)$ denotes some finite set of paths ending at b. (Note that this is a new and different use of the notation $\pi(b)$; the paths do not need to start at s.)

By what we just saw above, if the path P is an element of $\pi(b)$ and $\{Q_i\}$ is a finite set of paths such that $\bigwedge f(Q_i) \leq f(P)$, then P can be replaced in $\pi(b)$ by $\{Q_i\}$.

In Algorithm E (page 72), there is a choice made in the order in which the nodes are visited in the main loop. Let us fix such an order once and for all. (We will later see that there is a natural order to use.)

Let us modify Algorithm E to produce Algorithm F in Figure 5.6. Given a particular path

$$P : x_0 \to x_1 \to \ldots \to x_n,$$

Algorithm F computes $f(P)$. By comparing Algorithm E with Algorithm F, we see that if Algorithm F computes $f(P)$ in $n(P)$ iterations of the *while* loop, then after $n(P)$ iterations of the *while* loop in Algorithm E, we must have $m(b) \leq f(P)$.

```
begin
    m(s) ← λ;
    for each node x ∈ G − {s} do
        m(x) ← 1;
    end for;
    j ← 2;
    while j ≤ |P| do
        for each b ∈ G − {s} do
            if (b = x_j) then
                m(x_j) ← f_{x_{j−1}→x_j}(m(x_{j−1}));
                j ← j + 1;
            end if;
        end for;
    end while;
end
```

Figure 5.6: Algorithm F
Modified Algorithm E; Computes $f(P)$

Thus, if for each $b \in G$ there is a finite set $\pi(b)$ of paths ending at b (and not necessarily starting at s) such that

$$mop(b) = \bigwedge_{P \in \pi(b)} f(P)$$

and if N is the maximum number of iterations of the *while* loop in Algorithm F to compute $f(P)$ for all $P \in \cup\pi(b)$, then after N iterations of the *while* loop in Algorithm E, we have

$$m(b) \leq \bigwedge_{P \in \pi(b)} f(P) = mop(b)$$

for all $b \in G$, and therefore Algorithm E can be stopped after N iterations. (If in fact, the data-flow problem is distributive, then after N iterations, $mfp(b) \leq m(b) \leq mop(b) = mfp(b)$, so Algorithm E actually ends after N iterations.)

There is a useful class of data-flow problems for which a finite number of paths suffices to compute *mop*; this is the class of problems for which *mop* can be computed solely in terms of the simple paths. Since every flow graph is finite, there are only finitely many simple paths, and the analysis we have given above applies.

How can we characterize such data-flow problems? First, some notation: let us denote the sub-path from some node x_j to another node x_i of the path

$$P : x_0 \to x_1 \to \ldots \to x_n,$$

by $x_j \overset{*}{\to} x_i$, and let us write $f_{x_j \overset{*}{\to} x_i}$ for

$$f_{x_{i-1} \to x_i} \circ f_{x_{i-2} \to x_{i-1}} \circ \ldots \circ f_{x_j \to x_{j+1}}.$$

Thus, we have

$$f(P) = f_{x_0 \overset{*}{\to} x_n}(\alpha(x_0)) = f_{x_k \overset{*}{\to} x_n} \circ f_{x_0 \overset{*}{\to} x_k}(\alpha(x_0))$$

The *mop* solution is the meet of $f(P)$ over all such paths P which start at s. If $x_k = x_n$ (so the path has a cycle in it), then we would like to split the path into two paths, eliminating a cycle in the process; we will show below how to do this.

So suppose the path $x_0 \overset{*}{\to} x_n$ is split into $x_0 \overset{*}{\to} x_k$ and $x_k \overset{*}{\to} x_n$. We would like the expression for $f(P)$ to be an upper bound for

$$f_{x_k \overset{*}{\to} x_n}(\alpha(x_k)) \wedge f_{x_0 \overset{*}{\to} x_k}(\alpha(x_0)).$$

(If this is true, then as we have seen above, we can replace the path P by the two paths $x_0 \overset{*}{\to} x_k$ and $x_k \overset{*}{\to} x_n$ in computing *mop*.)

In particular, we want

$$f \circ g(\lambda) \geq f(\mathbf{1}) \wedge g(\lambda)$$

for all f and g in $\mathcal{F}$. Equivalently, setting $x = g(\lambda)$, we want

$$f(x) \geq f(\mathbf{1}) \wedge x. \tag{2}$$

It turns out that for distributive frameworks, this condition is sufficient:

5.11 Theorem *If $\langle L, \wedge, \mathcal{F} \rangle$ is a distributive framework satisfying* (2)*, then the* mop *solution of any instance of the framework can be found by forming the meet of $f(P)$ over all cycle-free paths.*

PROOF. Let

$$P : s = x_0 \to x_1 \to \ldots \to x_n = b$$

be a path from s to b which contains at least one cycle. Let j be the highest number such that x_j equals some x_k with $k > j$. Then $x_0 \overset{*}{\to} x_{j+1}$ and $x_{j+1} \overset{*}{\to} x_n$ together contain at least one less cycle than the

original path, and the second path is actually simple. We have

$$\begin{aligned} f(P) &= f_{x_{j+1} \xrightarrow{*} x_n} \circ f_{x_0 \xrightarrow{*} x_{j+1}}(\lambda) \\ &\geq f_{x_{j+1} \xrightarrow{*} x_n}(\mathbf{1}) \wedge f_{x_0 \xrightarrow{*} x_{j+1}}(\lambda) \end{aligned}$$

Hence we can replace any $P : s \xrightarrow{*} b$ which contains a cycle by a finite set of paths each of which has fewer cycles than P. By iterating this process, we can ultimately replace P by a finite set of cycle-free paths, and so *mop*(b) can be computed entirely in terms of cycle-free paths to b. □

Since there are only finitely many cycle-free paths, we see that for a distributive data-flow framework satisfying (2), *mop* can be calculated in terms of a finite number of paths. In fact, in this case, we can do even better, provided the graph is reducible.

For reducible graphs, the loop-connectedness number (see page 39) *lc*(G) is $\leq$ *dsl*(G), where *dsl*(G) is the derived sequence length of G, and in general is ≤ 3. Now suppose that in Algorithm F above, we walk each iteration of the nodes (in the *while* loop) in reverse postorder. Since for each path P, the nodes occur in reverse postorder except for the back arcs in P, we see that it will take at most *lc*(G) + 1 iterations of the *while* loop for algorithm F to compute $f(P)$ provided P is simple. (That is, it will take 1 iteration to reach the first back arc, and 1 more iteration to get past that arc and reach the next one, or the end of the path.)

For instance, if a path in some graph has the form

$$2 \to 3 \to 6 \to 16 \to 17 \to 20 \to 30 \to 12 \to 32 \to 40 \to 5 \to 9 \to 19$$

where the numbers are the reverse post-order numbers of the nodes in the path, then on the first iteration of the *while* loop in Algorithm F, nodes 2 through 30 will be visited. (The first back arc is $30 \to 12$.) On the next iteration, nodes 12 to 40 will be visited (up to the back arc $40 \to 5$). And then on the third iteration of the *while* loop, nodes 5 through 19 will be visited and $f(P)$ will have been computed. Here there are 2 back arcs in P, and it took 3 executions of the *while* loop to compute $f(P)$.

To sum up, for distributive data-flow problems satisfying (2), Algorithm E needs only *lc*(G) + 1 iterations of the *while* loop to compute *mop*. Of course, if we don't know *lc*(G), then we will need an extra iteration to confirm the "no change" test on the while loop, making the number of iterations *lc*(G) + 2. This result is due to Kam and Ullman[9].

Data-flow frameworks where the L is a semi-lattice of boolean arrays (or, equivalently, of sets), and where the functions $f \in \mathcal{F}$ are of the form

$$f(x) = (x \cap a) \cup b$$

are called *bit-vector frameworks*, because they are implemented using bit vectors.

5.12 Theorem (Hecht and Ullman[7]) *The* while *loop in Algorithm E is entered at most* lc(G) + 2 *times for bit-vector data-flow problems on reducible flow graphs, provided the nodes are visited in reverse postorder.*

PROOF. All we have to show is that the family of functions of the form

$$f(x) = (x \cap a) \cup b$$

satisfies (2). But

$$\begin{aligned} f(x) &= (x \cap a) \cup b \\ &\supseteq (x \cap a) \cup (x \cap b) \\ &= x \cap (a \cup b) \\ &= x \cap f(\mathbf{1}) \end{aligned}$$ □

Thus, the natural way to visit the nodes in Algorithm E is in reverse post-order; and if the graph is reducible, then we can actually get an effective bound on the running time of the algorithm when applied to bit-vector problems.

This kind of analysis can be extended to data-flow frameworks for which *mop* can be computed by considering paths containing at most k cycles. Again, the conditions only apply to distributive frameworks. Frameworks such as these are sometimes called *fast frameworks*. See for instance Tarjan's papers [21], [23], and [22].

More general conditions, not requiring distributivity, have also been considered by Graham and Wegman [4] and Rosen [15]. These conditions do not yield an efficient algorithm for *mop* [4]; but they do yield an efficient algorithm which computes a solution which is guaranteed to be between *mfp* and *mop*. These conditions, however, are not broad enough to include the problem of constant propagation considered earlier.

5.8 Algorithms Based on Reducibility

The algorithm we have been using up to this point consists of iterating over all the nodes of G a certain number of times. For this reason, it is called the iterative algorithm. For bit-vectoring problems where the flow graph is known to be reducible, there are some algorithms which potentially can save execution time, at the cost of managing more complicated data structures.

We will consider two such algorithms here. Both algorithms take their most natural form for reducible graphs, but both can be extended to apply to general flow graphs. Let us start, however, by assuming that G is reducible. Both these algorithms lean heavily on the fact that we can restrict ourselves to cycle-free paths when computing the data-flow solutions.

Since the algorithms are based on the derived sequence of a reducible flow graph, we cannot consider backward data-flow problems to be forward problems on the reverse flow graph—as we noted previously, the reverse graph of a reducible flow graph is not necessarily reducible. Thus separate algorithms are necessary for forward and backward problems.

Given a path Q in G of the form

$$Q : q_0 \to q_1 \to \ldots \to q_s,$$

let us write

$$f_Q = f_{q_{s-1} \to q_s} \circ f_{q_{s-2} \to q_{s-1}} \circ \ldots \circ f_{q_0 \to q_1}.$$

The algorithms will be based on the following lemma:

[4] In a negative direction, Kam and Ullman [10] have shown that there is no general algorithm which computes *mop* for *all* non-distributive problems.

5.13 Lemma *Let $\langle L, \wedge, \mathcal{F}\rangle$ and $\langle G, s\rangle$ be an instance of a distributive data-flow analysis framework satisfying DCC and (2). If all simple paths from s to x in G pass through a node k, then*

$$\mathrm{mop}(x) = \Bigg(\bigwedge_{\substack{\text{paths } Q \text{ from}\\ k \text{ to } x}} f_Q\Bigg)(\mathrm{mop}(k)).$$

Proof. Each simple path P from s to x corresponds uniquely to a pair of paths R from s to k, and Q from k to x, by the composition "P is R followed by Q"; and conversely. Correspondingly, we have $f_P = f_Q \circ f_R$. Hence by Lemma 5.8,

$$\begin{aligned}
mop(x) &= \bigwedge_{\substack{\text{paths } P \text{ from}\\ s \text{ to } x}} f_P(\lambda) \\
&= \bigwedge_{\substack{\text{paths } Q \text{ from}\\ k \text{ to } x}} \; \bigwedge_{\substack{\text{paths } R \text{ from}\\ s \text{ to } k}} f_Q \circ f_R(\lambda) \\
&= \bigwedge_{\substack{\text{paths } Q \text{ from}\\ k \text{ to } x}} f_Q\Bigg(\bigwedge_{\substack{\text{paths } R \text{ from}\\ s \text{ to } k}} f_R(\lambda)\Bigg) \\
&= \bigwedge_{\substack{\text{paths } Q \text{ from}\\ k \text{ to } x}} f_Q(mop(k)) \\
&= \Bigg(\bigwedge_{\substack{\text{paths } Q \text{ from}\\ k \text{ to } x}} f_Q\Bigg)(mop(k))
\end{aligned}$$

□

5.8.1 Allen and Cocke's Algorithm

This algorithm is essentially that presented in Allen and Cocke[3], although the notation is closer to that used in the Schwartz and Sharir algorithm, which we present subsequently.

$$G = G_0, G_1, \ldots, G_n = \{g\}$$

will be the derived sequence of G. The function ϕ_j will map each interval of G_j to its corresponding point in G_{j+1}. The function h_j, applied to an interval in G_j, returns the element of G_j which is the header of that interval.

Let us also define a function $\theta_j : G_j \to G$ as follows: θ_0 is just the identity function on G. If $j \geq 1$, then $\theta_j(x)$ is that element of G which represents the "ultimate header" of x. (For instance, in the example of Figure 3.11 on page 42, $\theta_3(C3) = C$.) Formally, we can define θ_j recursively as follows:

$$\theta_j(x) = \begin{cases} x & j = 0 \\ \theta_{j-1} h_{j-1} \phi_j^{-1}(x) & j > 0 \end{cases}$$

The functions $f_{x\to y}$ are assumed to have already been computed, and are denoted as $f_0(x, y)$.

These functions may seem complicated. The point to keep in mind is that they really come for free—they just represent the information in the data structure representing the flow graph and its derived sequence.

Let us make the convention that P is a path in G_j of the form

$$P : p_o \to p_1 \to \ldots \to p_r$$

and Q is a path in G of the form

$$Q : q_0 \to q_1 \to \ldots \to q_s.$$

ι will represent the identity function on the semi-lattice L.

The Forward Algorithm

The algorithm consists of two passes over the derived sequence. In the first pass, which proceeds from inner to outer intervals (that is, from G_0 to G_n), we compute at each stage (that is for each G_j) three sets of functions:

- For each arc $x \to y$ in G_j, $f_j(x, y)$, which will be computed recursively from the computations in G_{j-1}, will represent data flow from $\theta_j(x)$ to $\theta_j(y)$:

$$f_j(x,y) = \bigwedge_{\substack{\text{paths } Q \text{ from} \\ \theta_j(x) \text{ to } \theta_j(y)}} f_{q_{s-1}\to q_s} \circ f_{q_{s-2}\to q_{s-1}} \circ \ldots \circ f_{q_0\to q_1}$$

  (where $q_0 = \theta_j(x)$ and $q_s = \theta_j(y)$).

- For each interval $\langle K, k\rangle$ in G_j and each $x \in K$, $g_j(x)$ is the function which represents data flow from k to x within K:

$$\begin{aligned} g_j(x) &= \bigwedge_{\substack{\text{paths } P \text{ from} \\ k \text{ to } x}} f_j(p_{r-1}, p_r) \circ f_j(p_{r-2}, p_{r-1}) \circ \ldots \circ f_j(p_0, p_1) \\ &= \bigwedge_{\substack{\text{paths } Q \text{ from} \\ \theta_j(k) \text{ to } \theta_j(x)}} f_{q_{s-1}\to q_s} \circ f_{q_{s-2}\to q_{s-1}} \circ \ldots \circ f_{q_0\to q_1} \end{aligned}$$

  (where again $q_0 = \theta_j(x)$ and $q_s = \theta_j(y)$ and where $p_0 = k$ and $p_r = y$.)

  Thus for all $x \in K$,

$$g_j(x) = \bigwedge_{\substack{y\in \text{pred}_j(x) \\ y\in K}} f_j(y, x) \circ g_j(y).$$

- For each successor x of K, $F_j(K, x)$ is the function which represents data flow from k through K and out to x:

$$F_j(K, x) = \bigwedge_{\substack{y \in \mathrm{pred}_j(x) \\ y \in K}} f_j(y, x) \circ g_j(x).$$

  F_j will then be reinterpreted as f_{j+1} when G_{j+1} is processed.

In computing $g_j(x)$, we can restrict ourselves to cycle-free paths from k to x. This means, since K is an interval, that $g_j(x)$ can be computed in 1 pass over the nodes in K, except for paths with a back arc in them. The only such paths end at k: if $x = k$, we compute $g_j(x)$ separately by a final meet over paths from its predecessors.

The second pass over the derived sequence proceeds from outer to inner intervals. $m_n(g)$ is set equal to λ. Then for each interval $\langle K, k \rangle$ in G_j, $m_j(k)$ is set equal to $m_{j+1}(\phi_j(K))$. Finally, for each $x \in K$, $m_j(x)$ is set equal to $g_j(x)(m_j(k))$. By induction, we see that at each step of this pass, for each interval $\langle K, k \rangle$ in G_j,

- $m_j(k)$ is set equal to the *mop* solution at $\theta_j(k)$.
- Then, since for each $x \in K$, any cycle-free path from s to $\theta_j(x)$ must pass once through $\theta_j(k)$, Lemma 5.13 shows that $m_j(x)$ is set equal to the *mop* solution at $\theta_j(x)$.

Hence, at the end of the algorithm, which is in Figure 5.7, m_0 is the solution to the data-flow problem.

The Backward Algorithm

The modification of the Allen-Cocke algorithm for backwards data-flow propagation is due to Kennedy. His version, and the original references, can be found in [11].

The backwards problem is complicated by the fact that, although intervals are single-entry, they are not single-exit. Since we are following data-flow backwards through the flow graph, we have to take account of the fact that intervals can be entered at different points.

As usual, the exit node of the flow graph is denoted by e. Let us denote the image of e in G_j by e_j (so in particular, $e_0 = e$ and $e_n = g$). Further, let us adjoin a dummy "post-exit" node δ_j to each G_j, and a single edge $e_j \to \delta_j$. δ_j will not be thought of as being a member of G_j, and in particular, will not be a member of any interval of G_j. Its only function is to be a technical help in the algorithm. We set $f_{e_0 \to \delta_0} \leftarrow \iota$.

Again we make two passes over the derived sequence. In the first pass, which again proceeds from inner to outer intervals, (that is, from G_0 to G_n), we compute for each G_j three sets of functions:

- For each arc $x \to y$ in G_j, $f_j(x, y)$, which will be computed recursively from the computations in G_{j-1}, will represent data flow from $\theta_j(y)$ backwards to $\theta_j(x)$:

$$f_j(x, y) = \bigwedge_{\substack{\text{paths } Q \text{ from} \\ \theta_j(x) \text{ to } \theta_j(y)}} f_{q_0 \to q_1} \circ f_{q_1 \to q_2} \circ \ldots \circ f_{q_{s-1} \to q_s}$$

```
-- Step I: Inner-to-outer interval processing.
for j = 0 to n − 1 do
    if j ≠ 0 then
        -- f_0 is already defined.
        for each arc x → y in G_j do
            f_j(x, y) ← F_{j−1}(φ_j^{−1}(x), h_{j−1}(φ_j^{−1}(y)));
        end for;
    end if;
    for each interval ⟨K, k⟩ in G_j do/
        for each x ∈ K do
            initialize g_j(x) to the constant function 1;
        end for;
        repeat
            for each x ∈ K − {k} do
                g_j(x) ← ⋀_{y∈pred_j(x), y∈K} f_j(y, x) ∘ g_j(y);
            end for;
        until no change;
        g_j(k) ← ⋀_{y∈pred_j(k), y∈K} f(y, k) ∘ g_j(y);
        for each successor x of K do/
            F_j(K, x) ← ⋀_{y∈pred_j(x), y∈K} f_j(y, x) ∘ g_j(y);
        end for;
    end for;
end for;

-- Step II: Outer-to-inner interval processing.
m_n(g) ← λ;
for j = n − 1 to 0 step −1 do
    for each interval ⟨K, k⟩ in G_j do
        -- m_{j+1} is known.
        m_j(k) ← m_{j+1}(φ_j(K));
        for each x ∈ K − {k} do
            m_j(x) ← g_j(x)(m_j(k));
        end for;
    end for;
end for;
```

The *repeat* loop only has to be executed once provided the nodes in its *for* loop are visited in reverse post-order.

Figure 5.7: Allen and Cocke's Algorithm: Forward Data-Flow Propagation

(where $q_0 = \theta_j(x)$ and $q_s = \theta_j(y)$. Note the order in which the functions are composed to form each f_Q—this is backwards propagation.)

- For each interval $\langle K, k\rangle$ in G_j, each $x \in K$, and each $y \in \text{succ}(K)$, $g_j(x, y)$ represents the data flow backwards from the beginning of y (i.e. from the exit of K to y) back through K to x:

$$\begin{aligned} g_j(x,y) &= \bigwedge_{\substack{\text{paths } P \text{ from} \\ x \text{ to } y}} f_j(p_0,p_1) \circ f_j(p_1,p_2) \circ \ldots \circ f_j(p_{r-1},p_r) \\ &= \bigwedge_{\substack{\text{paths } Q \text{ from} \\ \theta_j(x) \text{ to } \theta_j(y)}} f_{q_0 \to q_1} \circ f_{q_1 \to q_2} \circ \ldots \circ f_{q_{s-1} \to q_s} \end{aligned}$$

  Thus for all $x \in K$,

$$g_j(x,y) = \bigwedge_{\substack{x \to w \\ w \in K \text{or } w = y}} f_j(x,w) \circ g_j(w,y)$$

  with the convention that $g_j(y, y) = \iota$.

- For each successor y of K,

$$F_j(K,y) = g_j(k,y).$$

In computing $g_j(x, y)$, if we visit the nodes x in K in reverse post-order, then at most two passes over the interval will be necessary, since any simple path from x to a successor y of K can contain at most one back arc (and this will be true iff the path contains the interval header k).

The second pass over the derived sequence proceeds from outer to inner intervals. We compute a function $\gamma_j : G_j \to \mathcal{F}$. If $x \in G_j$, $\gamma_j(x)$ will represent data flow backward from the exit e to $\theta_j(x)$ (that is, it will be the meet of all functions f_Q where Q runs over all paths from $\theta_j(x)$ to the exit).

γ_j is computed as follows: first, $\gamma_n(g)$ is set equal to $g_n(g, \delta_n)$. For each j (stepping down from $n - 1$ to 0), for each interval $\langle K, k\rangle$ in G_j, we first set $\gamma_j(k)$ to be γ_{j+1} of the image of K in G_{j+1}. Then again for each interval $\langle K, k\rangle$ in G_j, and for each node $x \in K - \{k\}$, we set

$$\gamma_j(x) \leftarrow \bigwedge_{y \in \text{succ}(K)} g_j(x,y) \circ \gamma_j(y).$$

We see by induction that for each interval $\langle K, k\rangle$ of G_j,

- $\gamma_j(k)$ is the meet of all functions f_Q where Q runs over all paths from $\theta_j(k)$ to the exit.
- Since for each $x \in K$, any cycle-free path from s to $\theta_j(x)$ must past once through $\theta_j(k)$, a minor variant of Lemma 5.13 shows that $\gamma_j(x)$ is the meet of all functions f_Q where Q runs over all paths from $\theta_j(x)$ to the exit.

Hence, at the end of the algorithm, the *mop* solution will be $m(x) = \gamma_0(x)(\lambda)$ for all $x \in G$.

The complete algorithm is given in Figure 5.8.

Kennedy's original algorithm did not use the fact that we knew $g_j(x, y)$ for all $x \in K$, and therefore computed γ_j as follows: after first computing γ_j for each interval header in G_j as above, for each $\langle K, k\rangle$ in G_j, the nodes $x \in K - \{k\}$ are visited in post-order. For each such node x, we set

$$\gamma_j(x) \leftarrow \bigwedge_{y \in \mathrm{succ}(x)} \begin{cases} f_j(x,y) \circ \gamma_j(y) & \text{if } y \in K \\ g_j(x,y) \circ \gamma_j(y) & \text{otherwise} \end{cases}$$

Since the nodes $x \in K$ are visited in post-order, each $y \in \mathrm{succ}(x)$ has either

1. been visited already in the interval; or
2. is the target of a back arc in the interval, in which case it is the interval header; or
3. is the target of an arc to another interval, in which case it is the header of another interval.

In each case, $\gamma_j(y)$ has already been assigned. Thus, one pass through each interval computes γ_j.

5.8.2 Schwartz and Sharir's Algorithm

One inefficiency in the algorithms as just presented, is that there may be nodes in one of the derived graphs which are unaffected by the next reduction, and so remain in the next derived graph. (See, for instance, Figure 3.10 on page 41.) Certainly there is no reason to visit these intervals again.

Schwartz and Sharir [16] noticed that the tree of reachunder sets created by Tarjan's algorithm for testing reducibility could be used instead of the tree reflecting the nesting of intervals of all orders to obtain a more efficient algorithm.

So instead of dealing with the derived sequence of G, we will deal with the sequence $\{\rho_j = \rho_j(k_j) : 1 \leq j \leq n\}$ defined as follows: For $1 \leq j < n$, ρ_j is the jth reachunder set produced by Tarjan's algorithm, and ρ_n is the limit graph produced by the algorithm. (Since G is assumed to be reducible, ρ_n is a DAG.) In each case, k_j is the header of ρ_j.

When a reachunder set ρ_j is identified by Tarjan's algorithm, it is collapsed into a point p_j, which becomes an element of the next set ρ_{j+1}. Instead of separate functions f_j, we will have one function $f(x, y)$, where x and y may be points in G or elements p_j, as appropriate. The forward algorithm is given in Figure 5.9.

For the backward algorithm, we only need one additional node δ, which, since it is not part of any reachunder set (or in fact, part of G), is never collapsed into any of the points $\{p_j\}$. We set $f(e, \delta) \leftarrow \iota$.

The backward algorithm is given in Figure 5.10.

5.8.3 Dealing With Non-Reducible Flow Graphs

The Schwartz-Sharir algorithm can be easily modified to handle non-reducible flow graphs, as follows: During the computation of reachunder sets in Tarjan's algorithm, the nodes k_j which are targets of back arcs are visited in reverse pre-order. If the reachunder set of k_j includes s (so we know that G is not reducible), then no action is taken other than marking k_j as "improper". Then the set k_{j+1} is computed as usual, and the algorithm proceeds. If any ρ_n contains an "improper" node, it is also marked as "improper".

```
-- Step I: Inner-to-outer interval processing.
for j = 0 to n − 1 do
    if j ≠ 1 then
        -- f_1 is already defined.
        for each arc x → y in G_j do
            f_j(x, y) ← F_{j−1}(φ_j^{−1}(x), h_{j−1}(φ_j^{−1}(y)));
        end for;
        f_j(e_j, δ_j) ← ι;
    end if;
    for each interval ⟨K, k⟩ in G_j do
        for each x ∈ K and y ∈ succ(K) do/
            Initialize g_j(x, y) to the constant function 1;
            Initialize g_j(y, y) to ι;
        end for;
        repeat
            for each x ∈ K do
                g_j(x, y) ← ⋀_{x→w, w∈K or w=y} f_j(x, w) ∘ g_j(w, y);
            end for;
        until no change;
        for each successor y of K do
            F_j(K, y) ← g_j(k, y);
        end for;
    end for;
end for;

-- Step II: Outer-to-inner interval processing.
γ_n(g) ← g_n(g, δ_n);
for j = n − 1 to 0 step −1 do
    for each interval ⟨K, k⟩ in G_j do
        γ_j(k) ← γ_{j+1}(φ_j(K));
    end for;
    for each interval ⟨K, k⟩ in G_j do
        for each x ∈ K − {k} do
            γ_j(x) ← ⋀_{y∈succ(K)} g_j(x, y) ∘ γ_j(y);
        end for;
    end for;
end for;

-- Step III: Final propagation step.
for each x ∈ G do
    m(x) ← γ_0(x)(λ);
end for;
```

The *repeat* loop only has to be executed twice provided the nodes in its *for* loop are visited in reverse post-order.

Figure 5.8: Allen and Cocke's Algorithm: Backward Data-Flow Propagation

```
-- Step I: Inner-to-outer interval processing.
for j = 0 to n − 1 do
    for each x ∈ ρ_j do
        initialize g_j(x) to the constant function 1;
    end for;
    repeat
        for each x ∈ ρ_j − {k_j} do
            g_j(x) ← ⋀_{y∈pred_j(x), y∈K} f(y, x) ∘ g_j(y);
        end for;
    until no change;
    g_j(k_j) ← ⋀_{y∈pred_j(k), y∈ρ_j} f(y, k_j) ∘ g_j(y);
    for each successor x of ρ_j do
        f(p_j, x) ← ⋀_{y∈pred_j(x), y∈ρ_j} f(y, x) ∘ g_j(y);
    end for;
    for each predecessor x of ρ_j do
        f(x, p_j) ← f(x, k_j);
    end for;
end for;

-- Step II: Outer-to-inner interval processing.
m_n(p_n) ← λ;
for j = n − 1 to 0 step −1 do
    -- m_{j+1} is known.
    m_j(k_j) ← m_{j+1}(p_j);
    for each x ∈ ρ_j − {k_j} do
        m_j(x) ← g_j(x)(m_j(k_j));
    end for;
end for;
```

If G is reducible, the *repeat* loop only has to be executed once provided the nodes in its *for* loop are visited in reverse post-order.

Figure 5.9: Schwartz and Sharir's Algorithm: Forward Data-Flow Propagation

```
-- Step I: Inner-to-outer interval processing.
for j = 0 to n − 1 do
    for each x ∈ ρ_j and y ∈ succ(ρ_j) do
        Initialize g_j(x, y) to the constant function 1;
        Initialize g_j(y, y) to ι;
    end for;
    repeat
        for each x ∈ K do/
            g_j(x, y) ← ⋀_{x→w, w∈K or w=y} f(x, w) ∘ g_j(w, y);
        end for;
    until no change;
    for each successor y of ρ_j do
        f(p_j, y) ← g_j(k, y);
    end for;
    for each predecessor x of ρ_j do
        f(x, p_j) ← f(x, k_j);
    end for;
end for;

-- Step II: Outer-to-inner interval processing.
γ_n(g) ← g_n(g, δ);
for j = n − 1 to 0 step −1 do
    γ_j(k_j) ← γ_{j+1}(p_j);
    for each x ∈ ρ_j − {k_j} do
        γ_j(x) ← ⋀_{y∈succ(ρ_j)} g_j(x, y) ∘ γ_j(y);
    end for;
end for;

-- Step III: Final propagation step.
for each x ∈ G do
    m(x) ← γ_0(x)(λ);
end for;
```

If G is reducible, the *repeat* loop only has to be executed twice provided the nodes in its *for* loop are visited in reverse post-order; and when $j = n$, it only has to be executed once (since ρ_n is known to be a DAG).

Figure 5.10: Schwartz and Sharir's Algorithm: Backward Data-Flow Propagation

The only change in the algorithms then that has to be made is that in Step I of each, the *repeat* loop for an improper ρ_j really has to be repeated until no change. That is, improper reachunder sets (or an improper terminal DAG) are handled by the iterative algorithm.

Bibliography

[1] Alfred V. Aho, John Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Code optimization and finite Church-Rosser systems. In Randall Rustin, editor, *Design and Optimization of Compilers*, pages 89–106. Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[3] F. E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, 1976.

[4] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[5] Matthew S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, New York, 1977.

[6] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. *SIAM Journal on Computing*, 1(2):188–202, June 1972.

[7] Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global flow problems. In *Conference Record of the First Annual ACM Symposium on Principles of Programming Languages, Boston (Mass.)*, pages 207–217, October 1973.

[8] Matthew S. Hecht and Jeffrey D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[9] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[10] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[11] Ken Kennedy. A Survey of Data Flow Analysis Techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[12] Gary Arlen Kildall. *Global Expression Optimization During Compilation.* PhD thesis, University of Washington, Seattle, Washington, May 1972.

[13] Gary Arlen Kildall. Global expression optimization during compilation. In *Conference Record of the First Annual ACM Symposium on Principles of Programming Languages, Boston (Mass.)*, pages 194–206, October 1973.

[14] Donald Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1(12):105–134, 1974.

[15] Barry K. Rosen. Monoids for rapid data flow analysis. *SIAM Journal on Computing*, 9(1):159–196, 1980.

[16] J. T. Schwartz and M. Sharir. A design for optimizations of the bitvectoring class. Technical Report 17, Courant Institute of Mathematical Sciences, Computer Science Department, September 1979.

[17] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computation and Mathematics with Applications*, 7(1):67–72, 1981.

[18] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3/4):141–153, 1981.

[19] Robert Endre Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[20] Robert Endre Tarjan. Testing flow graph reducibility. *J. Computer and System Sciences*, 9:355–365, 1974.

[21] Robert Endre Tarjan. Iterative algorithms for global flow analysis. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 71–102. Academic Press, 1976.

[22] Robert Endre Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

[23] Robert Endre Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.

[24] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

Index

ancestor, 2, 19
arc
    back, 3
    cross, 4
    forward, 3
    tree, 3
available expression, 59

backward equations, 63
boundary value equations, 62

chain, 45
    height, 45
child, 2, 19
collapsing a flow graph, 47
confluence equations, 63
consumes, 47
cycle, 19
cycle-free, 19

data flow analysis framework
    monotone, 69
data-flow problem
    distributive, 65
    monotone, 69
derived sequence, 29
derived sequence length, 39
descendant, 2, 19
descending chain condition, 72
distributive, 73
dominator, 20
    immediate, 21
    strong, 20
dominator tree, 21
dsl, *see* derived sequence length

entry node, 24

finite Church-Rosser transformation, 45
finite relation, 45
flow graph, 19
    derived, 28
    kernel, 51
    limit, 29
    reverse, 59
    trivial, 19
forward equations, 63
framework
    bit-vector, 81
    fast, 82
    monotone, 69
function
    distributive, 65
    monotonic, 68

head, 1
header, 24

interior, 19
interval, 24
interval nesting, 40
interval nesting tree, 42
irreducible, 29

kernel, 51

lattice, 68
lc, *see* loop-connectedness number
limit point, 45
live variable, 60
loop, 19
loop-connectedness number, 39
lower semi-lattice, 67
    complete, 68

maximal fixed-point solution, 72
meet-over-paths solution, 70
mfp, *see* maximal fixed-point solution

monotone data flow analysis framework
  instance, 70
mop, *see* meet-over-paths solution

nonreducible, 29

Path Lemma, 11
pre-interval, 24
predecessor, 2, 19
propagation equations, 63

reaching definition, 60
reachunder set, 53
reducible, 29
reduction ordering, 56
region, 24
reverse graph, 12
root of a strongly connected component, 13

self-loop, 19
simple, 19
source, 1
successor, 2, 19

tail, 1
target, 1
transfer equations, 63

very busy expression, 61