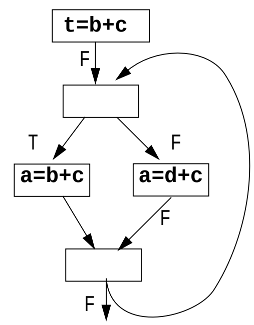
Very Busy Expressions and Loop Invariants

Very busy expressions are ideal candidates for invariant loop motion.

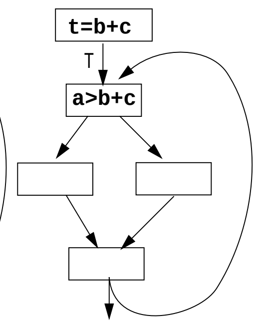
If an expression, invariant in a loop, is also very busy, we know it must be used in the future, and hence evaluation outside the loop must be worthwhile.


```
for (...) {
  if (...)
      a=b+c;
 else a=d+c;}
```

```
for (...) {
  if (a>b+c)
      x=1;
  else x=0;}
```



b+c is not very busy at loop entrance



b+c is very busy at loop entrance

Reaching Definitions

We have seen reaching definition analysis formulated as a set-valued problem. It can also be formulated on a per-definition basis.

That is, we ask "What blocks does a particular definition to v reach?"

This is a boolean-valued, forward flow data flow problem.

Initially, DefIn(b_0) = false.

For basic block b:

DefOut(b) =
  If the definition being analyzed is the last definition to v in b
  Then True
  Elsif any other definition to v occurs in b
  Then False
  Else DefIn(b)

The meet operation (to combine solutions) is:

$$\text{DefIn}(b) = \underset{p \in \text{Pred}(b)}{\text{OR}} \text{DefOut}(p)$$

To get all reaching definition, we do a series of single definition analyses.

Live Variable Analysis

This is a boolean-valued, backward flow data flow problem.

Initially, $LiveOut(b_{last})$ = false.

For basic block b:

LiveIn(b) =
 If the variable is used before it is defined in b
 Then True
 Elsif it is defined before it is used in b
 Then False
 Else LiveOut(b)

The meet operation (to combine solutions) is:

$$LiveOut(b) = \underset{s \in Succ(b)}{OR} LiveIn(s)$$

Bit Vectoring Data Flow Problems

The four data flow problems we have just reviewed all fit within a *single* framework.

Their solution values are Booleans (bits).

The meet operation is And or OR.

The transfer function is of the general form

$$Out(b) = (In(b) - Kill_b) \cup Gen_b$$

or

$$In(b) = (Out(b) - Kill_b) \cup Gen_b$$

where $Kill_b$ is true if a value is "killed" within b and Gen_b is true if a value is "generated" within b.

In Boolean terms:

$Out(b) = (In(b)$ AND Not $Kill_b)$ OR Gen_b

or

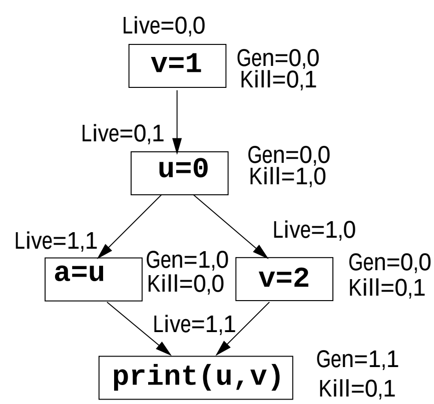
$In(b) = (Out(b)$ AND Not $Kill_b)$ OR Gen_b

An advantage of a bit vectoring data flow problem is that we can do a series of data flow problems "in parallel" using a bit vector.

Hence using ordinary word-level ANDs, ORs, and NOTs, we can solve 32 (or 64) problems simultaneously.

Example

Do live variable analysis for u and v, using a 2 bit vector:



We expect no variable to be live at the start of b_0. (Why?)

Reading Assignment

- Read pages 31-62 of “Automatic Program Optimization,” by Ron Cytron. (Linked from the class Web page.)

Depth-First Spanning Trees

Sometimes we want to “cover” the nodes of a control flow graph with an acyclic structure.

This allows us to visit nodes once, without worrying about cycles or infinite loops.

Also, a careful visitation order can approximate forward control flow (very useful in solving forward data flow problems).

A Depth-First Spanning Tree (DFST) is a tree structure that covers the nodes of a control flow graph, with the start node serving as root of the DFST.

Building a DFST

We will visit CFG nodes in depth-first order, keeping arcs if the visited node hasn’t be reached before.

To create a DFST, T, from a CFG, G:

1. T $\leftarrow$ empty tree
2. Mark all nodes in G as “unvisited.”
3. Call DF(start node)

DF (node) {

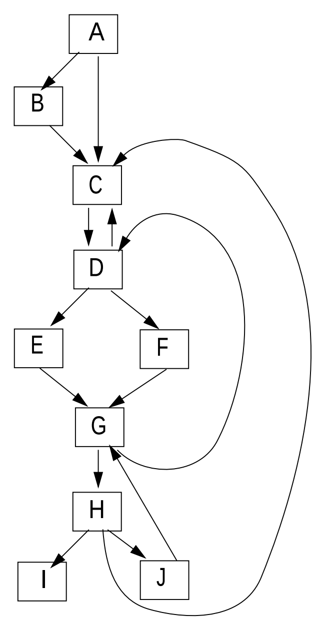
1. Mark node as visited.
2. For each successor, s, of node in G:

   If s is unvisited

   (a) Add node $\rightarrow$ s to T

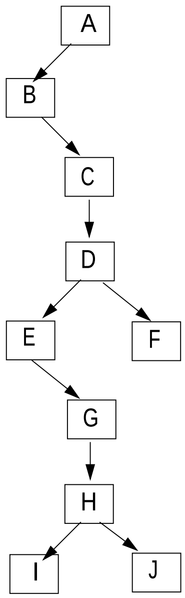
   (b) Call DF(s)

Example



Visit order is A, B, C, D, E, G, H, I, J, F

The DFST is



Categorizing Arcs using a DFST

Arcs in a CFG can be categorized by examining the corresponding DFST.

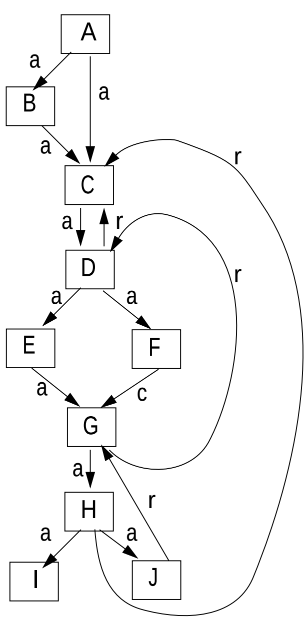
An arc A→B in a CFG is

(a) An *Advancing Edge* if B is a proper descendent of A in the DFST.

(b) A *Retreating Edge* if B is an ancestor of A in the DFST. (This includes the A→A case.)

(c) A *Cross Edge* if B is neither a descendent nor an ancestor of A in the DFST.

Example



Depth-First Order

Once we have a DFST, we can label nodes with a *Depth-First Ordering* (DFO).

Let i = the number of nodes in a CFG (= the number of nodes in its DFST).

DFO(node) {

  For (each successor s of node) do

    DFO(s);

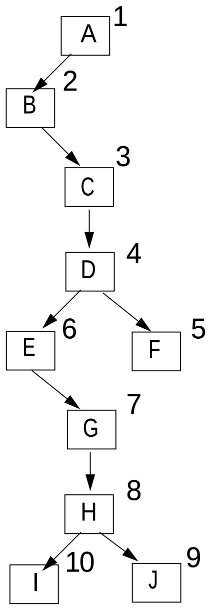
  Mark node with i;

  i--;

}

Example

The number of nodes = 10.



Application of Depth-First Ordering

- *Retreating edges* (a necessary component of loops) are easy to identify:
  a→b is a retreating edge if and only if dfo(b) ≤ dfo(a)
- A depth-first ordering in an excellent *visit order* for solving forward data flow problems. We want to visit nodes in essentially topological order, so that all predecessors of a node are visited (and evaluated) before the node itself is.

Dominators

A CFG node M *dominates* N (M dom N) if and only if *all* paths from the start node to N *must* pass through M.

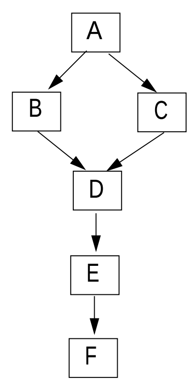
A node trivially dominates itself.

Thus (N dom N) is always true.

A CFG node M *strictly dominates* N (M sdom N) if and only if (M dom N) and M ≠ N.
A node can't strictly dominates itself.

Thus (N sdom N) is never true.

A CFG node may have many dominators.



Node F is dominated by F, E, D and A.

Immediate Dominators

If a CFG node has more than one dominator (which is common), there is always a unique "closest" dominator called its *immediate dominator*.

(M idom N) if and only if
(M sdom N) and
(P sdom N) $\Rightarrow$ (P dom M)

To see that an immediate dominator always exists (except for the start node) and is unique, assume that node N is strictly dominated by M_1, M_2, ..., M_p, $P \geq 2$.

By definition, M_1, ..., M_p must appear on *all* paths to N, including acyclic paths.

Look at the relative ordering among M_1 to M_p on some arbitrary acyclic path from the start node to N.

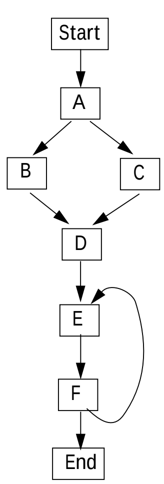
Assume that M_i is "last" on that path (and hence "nearest" to N).

If, on some other acyclic path, $M_j \neq M_i$ is last, then we can shorten this second path by going directly from M_i to N without touching any more of the M_1 to M_p nodes.

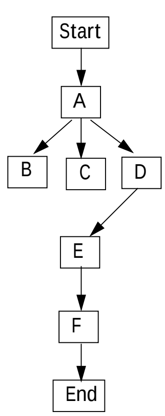
But, this totally removes M_j from the path, contradicting the assumption that (M_j sdom N).

Dominator Trees

Using immediate dominators, we can create a *dominator tree* in which A$\rightarrow$B in the dominator tree if and only if (A idom B).

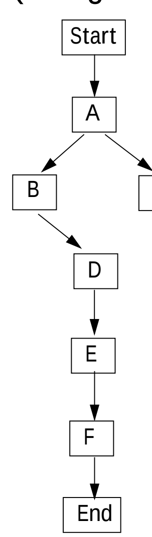


Control Flow Graph

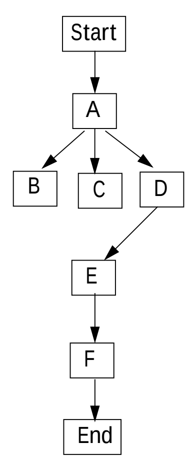


Dominator Tree

Note that the Dominator Tree of a CFG and its DFST are distinct trees (though they have the same nodes).



Depth-First Spanning Tree



Dominator Tree

A Dominator Tree is a compact and convenient representation of both the dom and idom relations.

A node in a Dominator Tree dominates all its descendents in the tree, and immediately dominates all its children.

Computing Dominators

Dominators can be computed as a Set-valued Forward Data Flow Problem.

If a node N dominates all of node M's predecessors, then N appears on all paths to M. Hence (N dom M).

Similarly, if M *doesn't* dominate all of M's predecessors, then there is a path to M that doesn't include M. Hence $\neg$(N dom M).

These observations give us a "data flow equation" for dominator sets:

$$\mathrm{dom}(N) = \{N\} \cup \bigcap_{M \in \mathrm{Pred}(N)} \mathrm{dom}(M)$$

The analysis domain is the lattice of all subsets of nodes. Top is the set of all nodes; bottom is the empty set. The ordering relation is subset.

The meet operation is intersection.

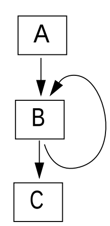
The Initial Condition is that
$\mathrm{DomIn}(b_0) = \phi$

$$\mathrm{DomIn}(b) = \bigcap_{c \in \mathrm{Pred}(b)} \mathrm{DomOut}(c)$$

$$\mathrm{DomOut}(b) = \mathrm{DomIn}(b) \cup \{b\}$$

Loops Require Care

Loops in the Control Flow Graph induce circularities in the Data Flow equations for Dominators. In



we have the rule dom(B) =
DomOut(B) =
DomIn(B) ∪ {B} =
{B} ∪ ($\underline{\mathrm{DomOut}(B)}$ $\cap$ DomOut(A))

If we choose DomOut(B) = ϕ initially, we get DomOut(B) =
{B} ∪ (ϕ $\cap$ DomOut(A)) = {B}
which is *wrong*.

Instead, we should use the Universal Set (of all nodes) which is the *identity* for $\cap$.

Then we get DomOut(B) =
$\{B\} \cup (\{\text{all nodes}\} \cap \text{DomOut}(A))$ =
$\{B\} \cup \text{DomOut}(A)$
which is correct.