Control Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Outline of the Lecture

- Why control flow analysis?
- Dominators and natural loops
- Intervals and reducibility
- $T_1 - T_2$ transformations and graph reduction
- Regions

Why Control-Flow Analysis?

- Control-flow analysis (CFA) helps us to understand the structure of control-flow graphs (CFG)
- To determine the loop structure of CFGs
- Formulation of conditions for code motion use dominator information, which is obtained by CFA
- Construction of the static single assignment form (SSA) requires dominance frontier information from CFA
- It is possible to use interval structure obtained from CFA to carry out data-flow analysis
- Finding Control dependence, which is needed in parallelization, requires CFA

Dominators

- We say that a node d in a flow graph *dominates* node n, written d *dom* n, if every path from the initial node of the flow graph to n goes through d
- Initial node is the root, and each node dominates only its descendents in the dominator tree (including itself)
- The node x *strictly dominates* y, if x dominates y and $x \neq y$
- x is the *immediate dominator* of y (denoted $idom(y)$), if x is the closest strict dominator of y
- A *dominator tree* shows all the immediate dominator relationships
- Principle of the dominator algorithm
  - If $p_1, p_2, ..., p_k$, are all the predecessors of n, and $d \neq n$, then d *dom* n, iff d *dom* p_i for each i

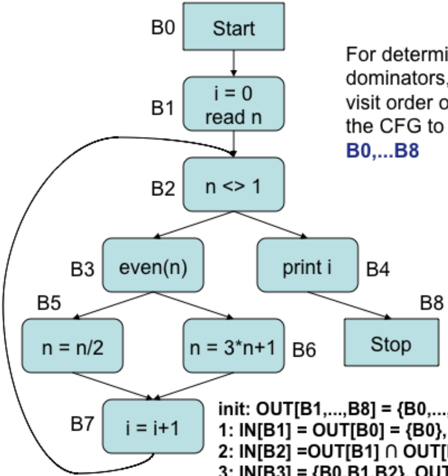
An Algorithm for finding Dominators

- $D(n) = OUT[n]$ for all n in N (the set of nodes in the flow graph), after the following algorithm terminates
- { /* n_0 = initial node; N = set of all nodes; */
  $OUT[n_0] = \{n_0\}$;
  for n in $N - \{n_0\}$ do $OUT[n] = N$;
  while (*changes to any* $OUT[n]$ *or* $IN[n]$ *occur*) do
  for n in $N - \{n_0\}$ do

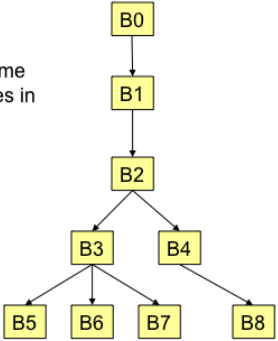
$$
\begin{aligned}
IN[n] &= \bigcap_{P \text{ a predecessor of } n} OUT[P]; \\
OUT[n] &= \{n\} \cup IN[n]
\end{aligned}
$$

}

Dominator Example

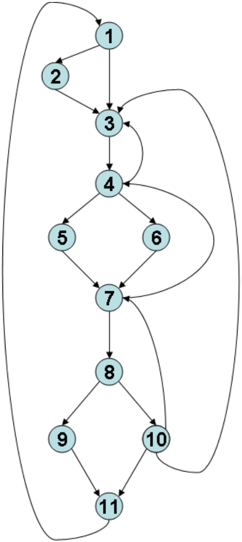


For determining dominators, assume visit order of nodes in the CFG to be **B0,...B8**

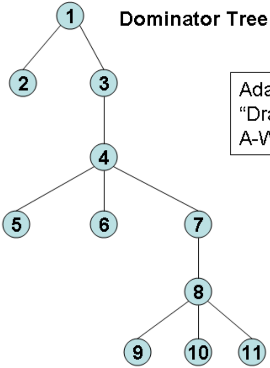


init: OUT[B1,...,B8] = {B0,...,B8}, OUT[B0] = {B0}
1: IN[B1] = OUT[B0] = {B0}, OUT[B1] = {B0,B1}
2: IN[B2] =OUT[B1] ∩ OUT[B7] = {B0,B1} , OUT[B2] = {B0,B1,B2}
3: IN[B3] = {B0,B1,B2}, OUT[B3] = {B0,B1,B2,B3}
IN[B4] = {B0,B1,B2}, OUT[B4] = {B0,B1,B2,B4}
4: IN[B5] = {B0,B1,B2,B3} = IN[B6], OUT[B5] = {B0,B1,B2,B3,B5}
OUT[B6] = {B0,B1,B2,B3,B6}, OUT[B8] = {B0,B1,B2,B4,B8}
5: IN[B7] = OUT[B5] ∩ OUT[B6] = {B0,B1,B2,B3}
OUT[B7] = {B0,B1,B2,B3,B7}

Dominators, Back Edges, and Natural Loops



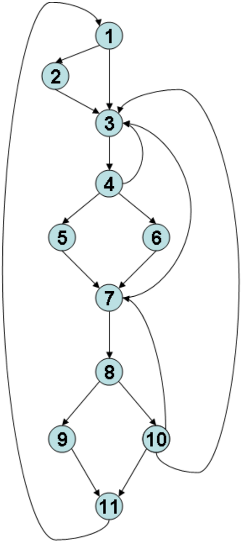
Flow Graph



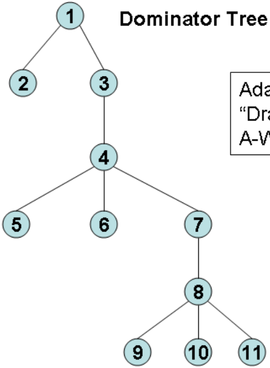
Dominator Tree

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

Dominators, Back Edges, and Natural Loops



Flow Graph



Dominator Tree

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

Dominators and Natural Loops

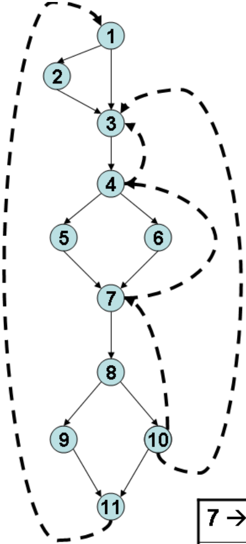
- Edges whose heads dominate their tails are called *back edges* ($a \rightarrow b: \ b = head, \ a = tail$)
- Given a back edge $n \rightarrow d$
  - The *natural loop* of the edge is d plus the set of nodes that can reach n without going through d
  - d is the header of the loop
    - A single entry point to the loop that dominates all nodes in the loop
    - Atleast one path back to the header exists (so that the loop can be iterated)

Algorithm for finding the Natural Loop of a Back Edge

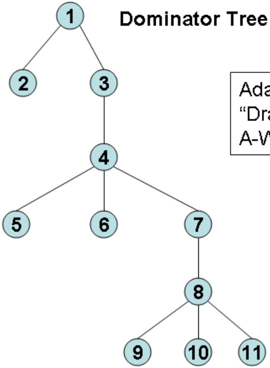
```
/* The back edge under consideration is n → d /*
{ stack = empty; loop = {d};
  /* This ensures that we do not look at predecessors of d */
   insert(n);
   while (stack is not empty) do {
      pop(m, stack);
      for each predecessor p of m do insert(p);
   }
}

   procedure insert(m) {
      if m ∉ loop then {
         loop = loop ∪ {m};
         push(m, stack);
      }
   }
```

Dominators, Back Edges, and Natural Loops



Flow Graph



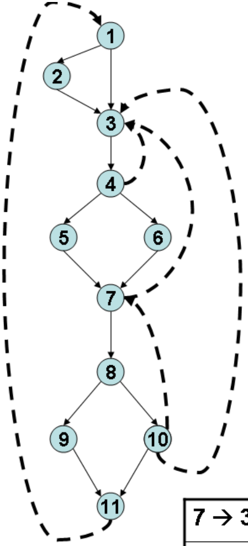
Dominator Tree

Adapted from the “Dragon Book”, A-W, 1986

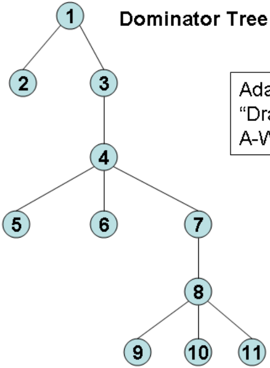
Back edges and their natural loops

7 → 4	10→7	4→3	10→3	11→1
{4,5,6,7,8,10}	{7,8,10}	{3,4,5,6,7,8,10}	{3,4,5,6,7,8,10}	{1,2,3,4,5,6,7,8,9,10,11}

Dominators, Back Edges, and Natural Loops



Flow Graph



Dominator Tree

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

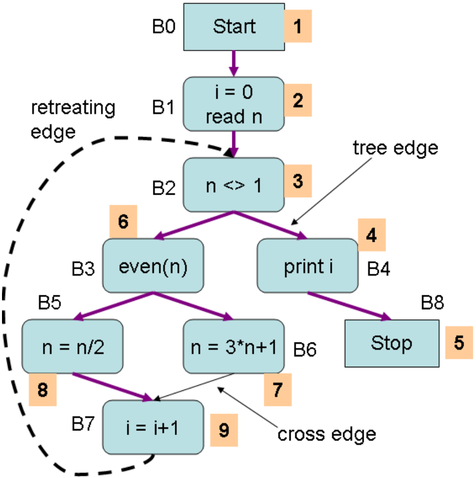
Back edges and their natural loops

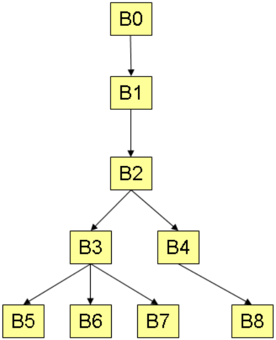
7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6,7,8, 10}	{7,8,10}	{3,4}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Depth-First Numbering of Nodes in a CFG

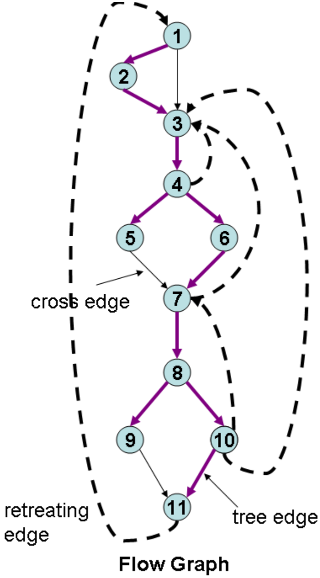
```
void dfs-num(int n) {
    mark node n "visited";
    for each node s adjacent to n do {
        if s is "unvisited" {
            add edge n →s to dfs tree T;
            dfs-num(s);
        }
     depth-first-num[n] = i ; i-- ;
}
// Main program
{  T = empty; mark all nodes of CFG as "unvisited";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

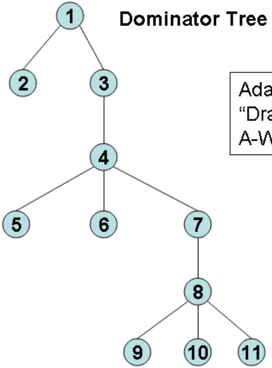
Depth-First Numbering Example 1





Depth-First Numbering Example 2





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

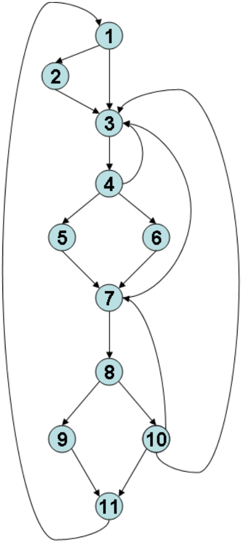
Nodes of the CFG show the DF-numbering

Reducibility

A flow graph G is reducible iff

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

Reducibility - Example 1



Flow Graph

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

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

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

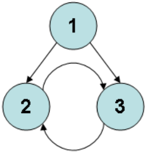
Reducible graph.

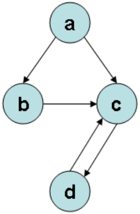
Reducibility - Example 2

Irreducible graph, no back edge.

Either 2 → 3 or 3 → 2 is a retreating edge in a depth-first search tree.

The graph is cyclic, not a DAG.





d → c is a back edge.

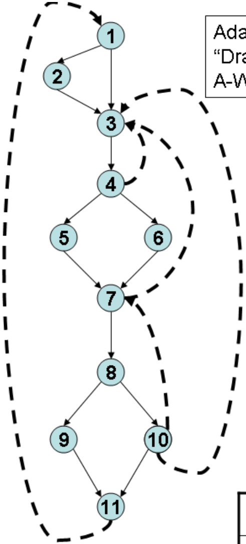
Other edges form a DAG in which each node is reachable from the node a.

Reducible graph.

Inner Loops

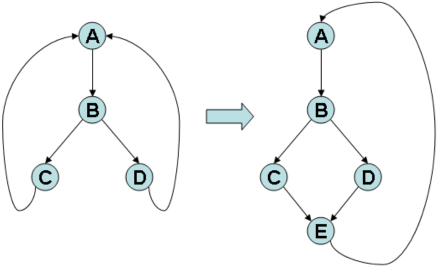
- Unless two loops have the same header, they are either disjoint or one is nested within the other
- Nesting is checked by testing whether the set of nodes of a loop A is a subset of the set of nodes of another loop B
- Similarly, two loops are disjoint if their sets of nodes are disjoint
- When two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

Inner Loops and Loops with the same header



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

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

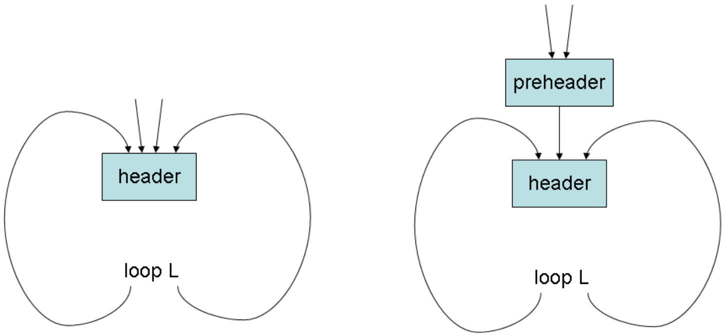


E is a dummy node

Back edges and their natural loops

7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6, 7,8,10}	{7,8,10}	{3,4}	{3,4,5,6, 7,8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

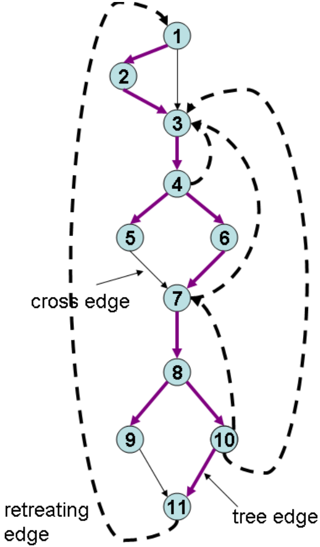
Preheader



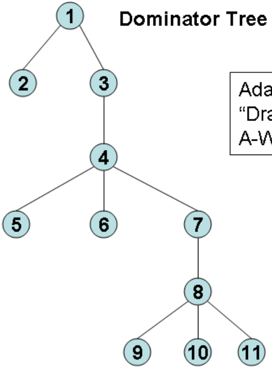
Depth of a Flow Graph and Convergence of DFA

- Given a depth-first spanning tree of a CFG, the largest number of retreating edges on any cycle-free path is the *depth* of the CFG
- The number of passes needed for convergence of the solution to a forward DFA problem is (1 + depth of CFG)
- One more pass is needed to determine *no change*, and hence the bound is actually (2 + depth of CFG)
- This bound can be actually met if we traverse the CFG using the *depth-first numbering* of the nodes
- For a backward DFA, the same bound holds, but we must consider the reverse of the depth-first numbering of nodes
- Any other order will still produce the correct solution, but the number of passes may be more

Depth of a CFG - Example 1



Flow Graph

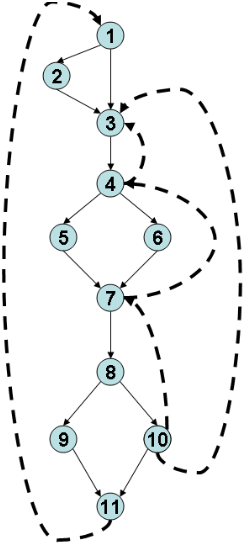


Adapted from the "Dragon Book", A-W, 1986

Nodes of the CFG show the DF-numbering

Depth of the CFG = 2 (10-7-3)

Depth of a CFG - Example 2



Flow Graph

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

Depth of the CFG = 3 (10-7-4-3)

Intervals

- Intervals have a header node that dominates all nodes in the interval
- Given a flow graph G with initial node n_0, and a node n of G, the interval with header n, denoted $I(n)$ is defined as follows
    1. n is in $I(n)$
    2. If all the predecessors of some node $m \neq n_0$ are in $I(n)$, then m is in $I(n)$
    3. Nothing else is in $I(n)$
- Constructing $I(n)$

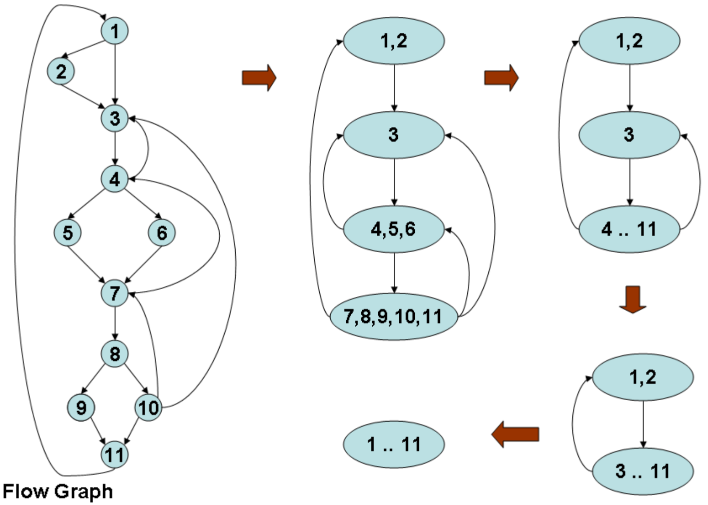
$I(n) := \{n\}$;
while (there exists a node $m \neq n_0$, all of whose predecessors are in $I(n)$) do $I(n) := I(n) \cup \{m\}$;

Partitioning a Flow Graph into Disjoint Intervals

```
Mark all nodes as "unselected";
Construct I(n0); /* n0 is the header of I(n0) */
Mark all the nodes in I(n0) as "selected";
while (there is a node m, not yet marked "selected",
      but with a selected predecessor) do {
      Construct I(m);/* m is the header of I(m) */
      Mark all nodes in I(m) as "selected";
}
```

Note: The order in which interval headers are picked does not alter the final partition

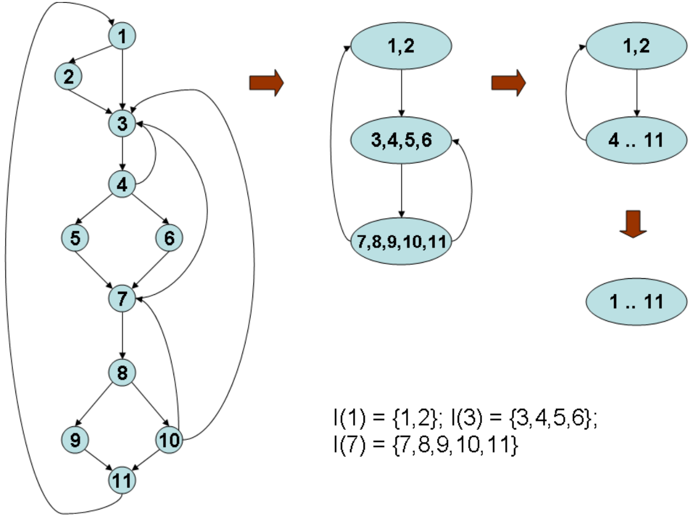
Intervals and Reducibility - 1



I(1) = {1,2}; I(3) = {3}
I(4) = {4,5,6}; I(7) = {7,8,9,10,11}

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

Intervals and Reducibility - 2



I(1) = {1,2}; I(3) = {3,4,5,6};
I(7) = {7,8,9,10,11}

Flow Graph

Interval Graphs

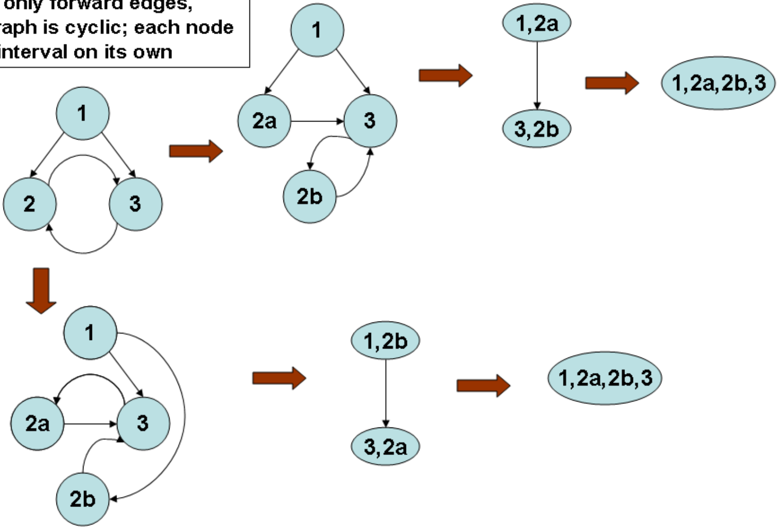
- Intervals correspond to nodes
- Interval containing n_0 is the initial node of $I(G)$
- If there is an edge from a node in interval $I(m)$ to the header of the interval $I(n)$, in G, then there is an edge from $I(m)$ to $I(n)$ in $I(G)$
- We make intervals in interval graphs and reduce them further
- Finally, we reach a *limit flow graph*, which cannot be reduced further
- *A flow graph is reducible iff its limit flow graph is a single node*

Node Splitting

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

Node Splitting Example

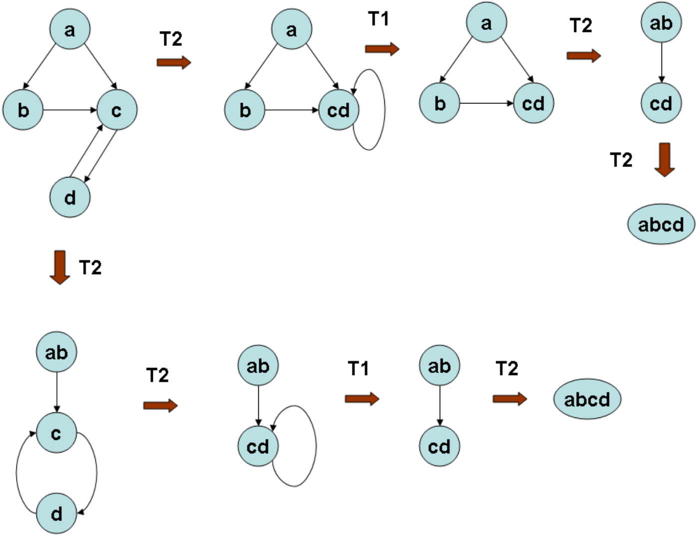
Irreducible graph; no back edge, only forward edges, but graph is cyclic; each node is an interval on its own



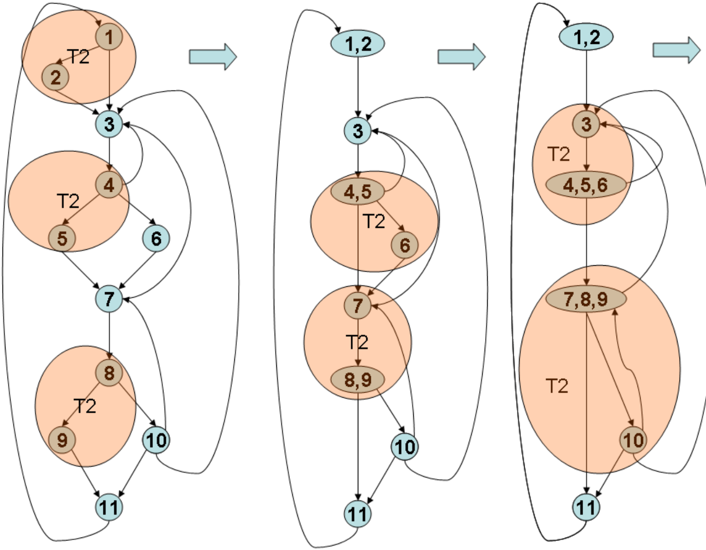
$T_1 - T_2$ Transformations and Graph Reduction

- **Transformation** T_1: If n is a node with a loop, *i.e.*, an edge $n \rightarrow n$ exists, then delete that edge
- **Transformation** T_2: If there is a node n, not the initial node, that has a unique predecessor m, then m may *consume* n by deleting n and making all successors of n (including n, possibly) be successors of m
- By applying the transformations T_1 and T_2 repeatedly in any order, we reach the limit flow graph
- Node splitting may be necessary as in the case of interval graph reduction

Example of $T_1 - T_2$ Reduction

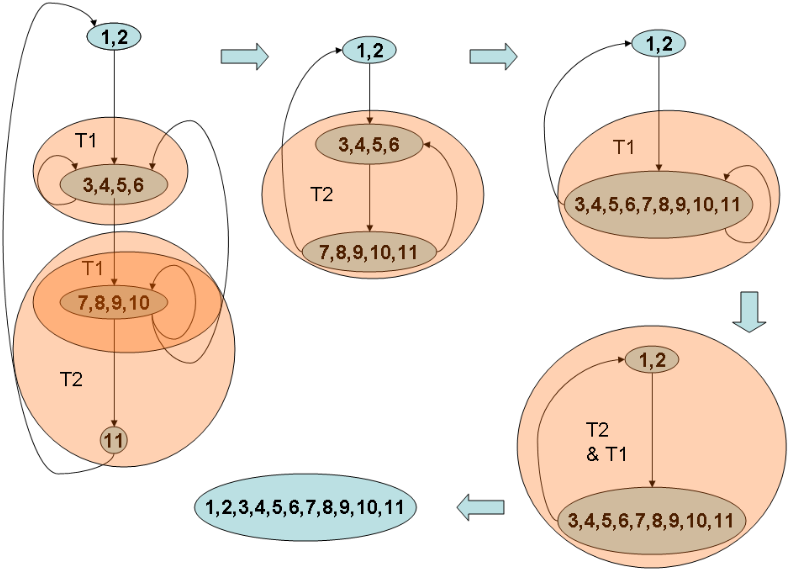


Example of $T_1 - T_2$ Reduction



Flow Graph

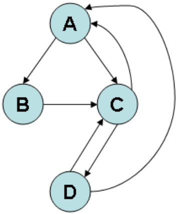
Example of $T_1 - T_2$ Reduction

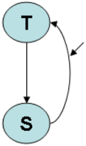


Regions

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

Region Example





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