The Lengauer Tarjan Algorithm
for Computing the
Immediate Dominator Tree
of a Flowgraph

by

Martin Richards
mr@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~mr10

Revised: Fri Jul 28 15:07:06 BST 2017
DFS of the Flowgraph
Processing Node 7

- Node being processed
- Immediate predecessors of 7
- DFS Tree edges to nodes > 7
Processing Node 2

Node being processed
Immediate predecessor of 2
DFS Tree edges to nodes > 2
Nodes with Semidominator 18

DFS Tree edges to nodes > 18
Nodes with Semidominator 2

Node being processed

Nodes with semidominator 2

DFS Tree edges to nodes > 2
Nodes with Semidominator 1

Node being processed

Nodes with semidominator 1

DFS Tree edges to nodes > 1
Final Phase

Nodes changed in final pass

Diagram showing the final phase with nodes and edges.
Step 1: Initialisation

Vertices in depth first search discovery order from 1 to n.

For each vertex v from 1 to n set:

\[
\begin{align*}
\text{parent}[v] & := \text{DFS tree parent of } v \\
\text{succs}[v] & := \text{the given list of successors} \\
\text{preds}[v] & := \text{list of predecessors} \\
\text{semi}[v] & := v \\
\text{idom}[v] & := 0 \\
\text{ancestor}[v] & := 0 \\
\text{best}[v] & := v \\
\text{bucket}[v] & := 0
\end{align*}
\]

Note that indirection in BCPL normally uses expressions such as `parent![v]`, but for compatibility with other languages `parent[v]` is also allowed.
FOR $w = n$ TO 2 BY -1 DO
{ LET $p = \text{parent}[w]$

step2: FOR each $v$ in $\text{preds}[w]$ DO
{ LET $u = \text{EVAL}(v)$
    IF $\text{semi}[w] > \text{semi}[u]$ DO
      $\text{semi}[w] := \text{semi}[u]$
    }
add $w$ to $\text{bucket}[\text{semi}[w]]$
LINK($p$, $w$)

step3: FOR each $v$ in $\text{bucket}[p]$
{ LET $u = \text{EVAL}(v)$
    $\text{idom}[v] := \text{semi}[u]<\text{semi}[v] \rightarrow u$, $p$
} 
$\text{bucket}[p] := 0$
}

step4: FOR $w = 2$ TO $n$ DO
UNLESS $\text{idom}[w] = \text{semi}[w]$ DO
  $\text{idom}[w] := \text{idom}[\text{idom}[w]]$
$idom[1] := 0$
Very Simple LINK and EVAL

LET LINK(v, w) BE \text{ancestor}[w] := v

LET EVAL(v) = VALOF
\{ LET a = \text{ancestor}[v]

\text{WHILE} \text{ancestor}[a] \text{ DO}
\{ IF \text{semi}[v] > \text{semi}[a] \text{ DO } v := a
 \quad a := \text{ancestor}[a]
\}

// v is now the vertex
// with earliest semidominator
// of any in the ancestor chain.

RESULT IS v
\}
Simple LINK and EVAL

LET LINK(v, w) BE ancestor[w] := v

LET EVAL(v) = VALOF
{ UNLESS ancestor[v] RESULTIS v
  COMPRESS(v)
  RESULTIS best[v]
}

AND COMPRESS(v) BE
{ LET a = ancestor[v]

  UNLESS ancestor[a] RETURN

  COMPRESS(a)

  IF semi[best[v]] > semi[best[a]] DO
    best[v] := best[a]
    ancestor[v] := ancestor[a]
  }

After calling EVAL(17)
Sophisticated EVAL

This version of EVAL calls COMPRESS to perform the following optimisation of the ancestor chain wherever possible.

If there is an ancestor link from x to y and one from y to z, then the link from x to y is replaced by one from x to z updating the best field of x if necessary.

The effect of this optimisation is to modify the ancestor links so that the ancestor chain length is less than 2 for every vertex in the original chain. This clearly increases the efficiency of later calls of EVAL.

LET EVAL(v) = VALOF
{ UNLESS ancestor[v] RESULTIS best[v]
  COMPRESS(v)
  RESULTIS semi[best[ancestor[v]]] < semi[best[v]] ->
    best[ancestor[v]], best[v]
}
Sophisticated LINK

LET LINK(v, w) BE
{ LET s = w

{ LET cs = child[s]  // cs = child(s)
  LET bcs = cs -> best[cs], 0  // bcs = best(child(s))

  TEST cs &
  semi[best[w]] < semi[bcs]  // bcs=0 only when cs=0
  THEN { // Combine the first two trees in the child chain, // making the larger one the combined root.
    LET ccs = child[cs]  // ccs = child(child(s))
    LET ss = size[s]  // sc = size(s)
    LET scs = size[cs]  // scs = size(child(s)
    LET sccs = ccs -> size[ccs], 0  // sccs=size(child(child(s))
    TEST ss-scs >= scs-sccs // Compare first two tree sizes.
    THEN { ancestor[cs] := s // The first is larger or equal.
      child[s] := ccs
      ELSE { size[cs] := ss // The second is larger.
        ancestor[s] := cs
        s := cs
      }
  }
  ELSE { BREAK }
} REPEAT

// Now combine the two forests giving the combination the // child chain of the smaller forest. The other child chain is // then collapsed, giving all its trees ancestor links to v.
best!s := best!w
IF size[v]<size[w] DO { LET t = s; s := child[v]; child[v] := t }
size[v] := size[v] + size[w]
WHILE s DO { ancestor[s] := v; s := child[s] }
}
Balanced Trees

The number next to each vertex $v$ in the following diagram is $\text{semi}[\text{best}[v]]$ and, in the child chains, these are non decreasing. Note that child links are like reversed ancestor links but with this monotonicity property.

![Diagram showing balanced tree structure with links and values]

LINK($v$, $w$)
Balanced Trees 1

Just before LINK is called the best value of q may have been reduced possibly requiring its child chain to be modified to reinstate its monotonicity.
Balanced Trees 3

LINK(v, w)

Subtree ancestor link

Child link
Balanced Trees 4a

The following is the result if the forest rooted at v had fewer vertices than the forest that was rooted at w.
Balanced Trees 4b

The following is the result if the forest rooted at $v$ had the same number or more vertices than the forest that was rooted at $w$.

LINK($v$, $w$)
Experimental Results

Results from running the BCPL program `bcplprogs/dom/lt.b` which applies the three variants of the algorithm to random graphs.

<table>
<thead>
<tr>
<th>Random Graph</th>
<th>Cintcode Instruction Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>Edges</td>
</tr>
<tr>
<td>1000</td>
<td>1500</td>
</tr>
<tr>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>1000</td>
<td>2500</td>
</tr>
<tr>
<td>1000</td>
<td>3000</td>
</tr>
<tr>
<td>1000</td>
<td>5000</td>
</tr>
<tr>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>10000</td>
<td>50000</td>
</tr>
<tr>
<td>10000</td>
<td>100000</td>
</tr>
<tr>
<td>100000</td>
<td>400000</td>
</tr>
<tr>
<td>100000</td>
<td>123289</td>
</tr>
</tbody>
</table>