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

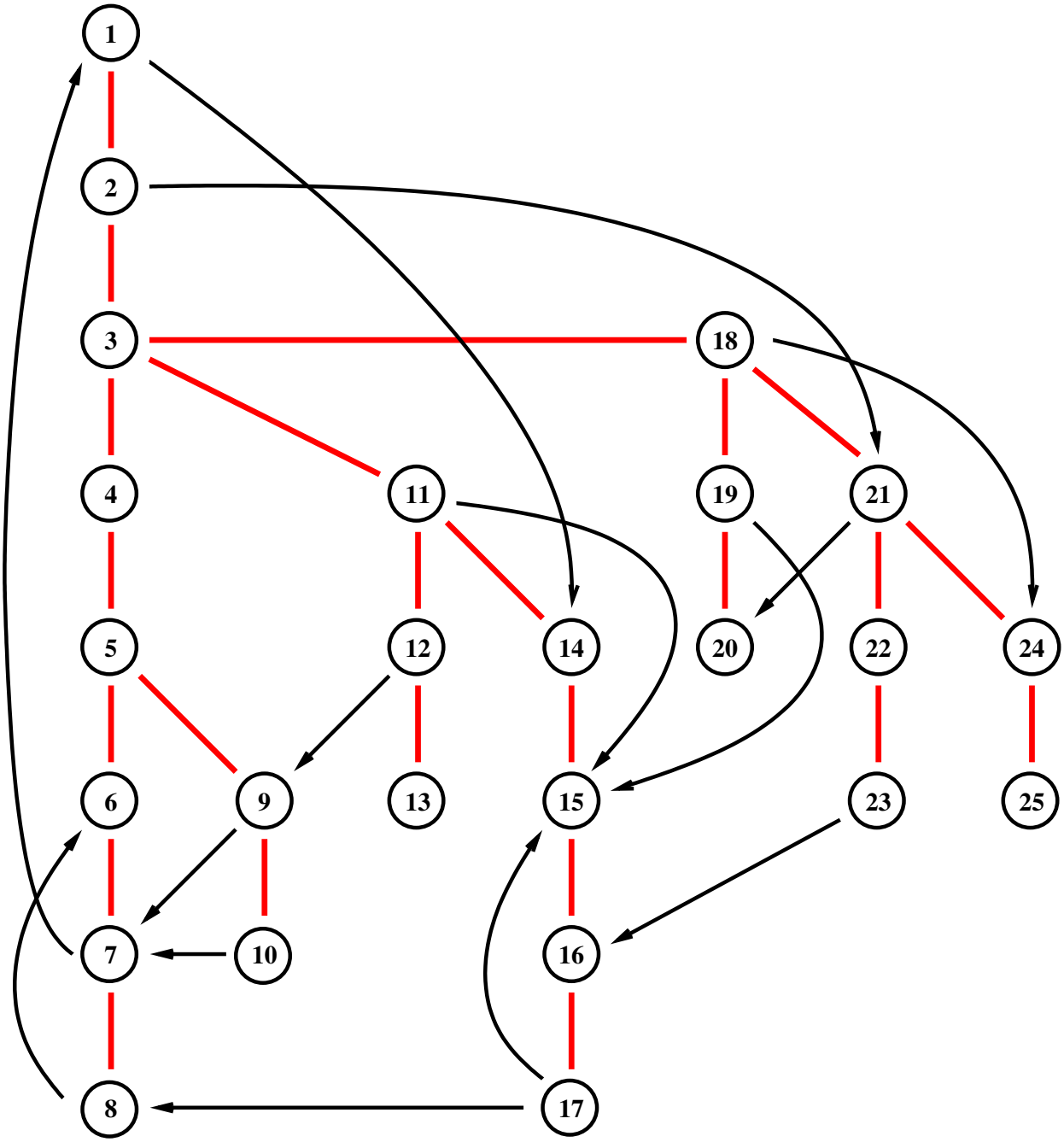
by

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

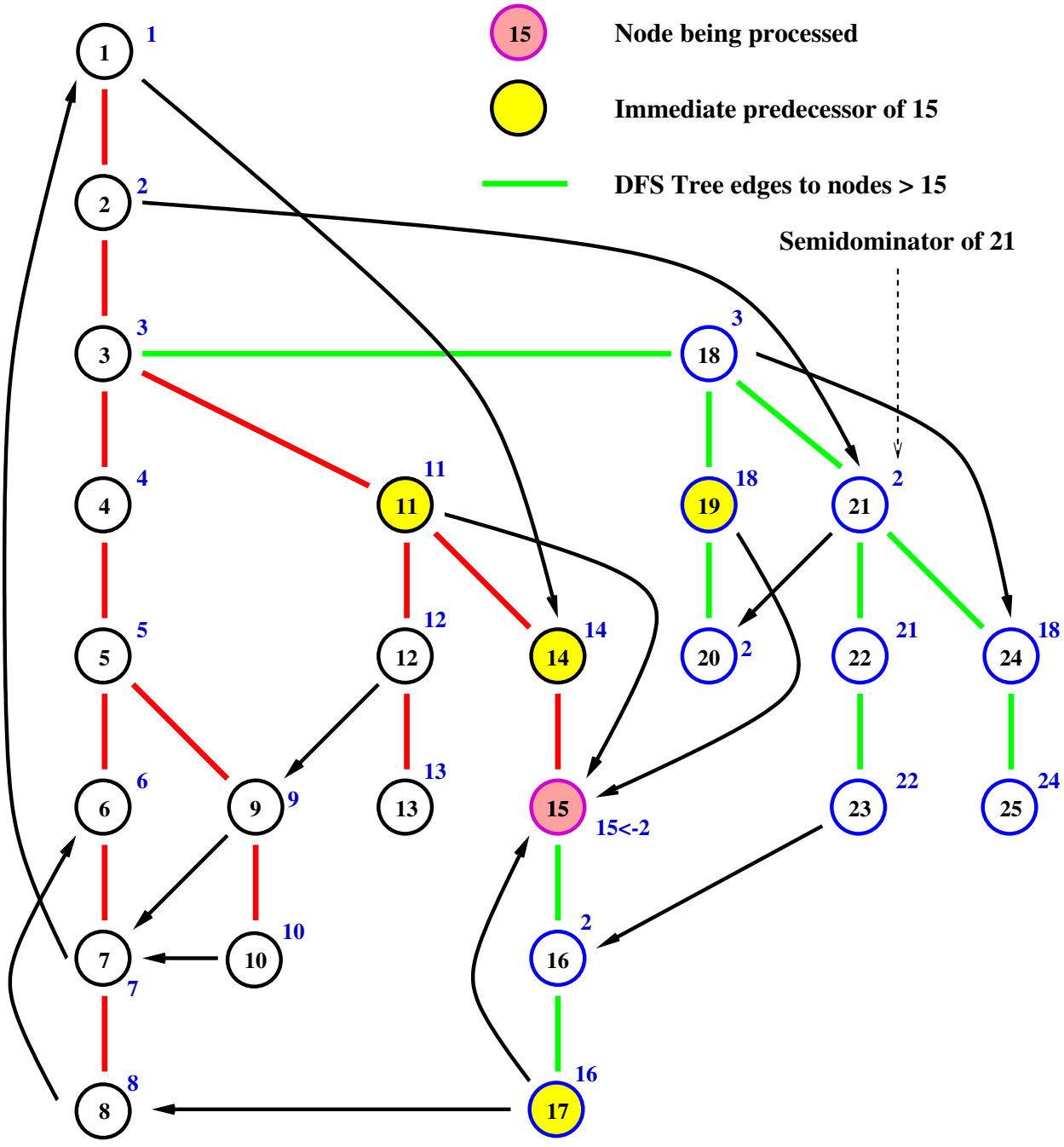
Revised: Fri Jul 28 15:07:06 BST 2017

University Computer Laboratory
JJ Thomson Avenue
Pembroke Street
Cambridge, CB3 0FD

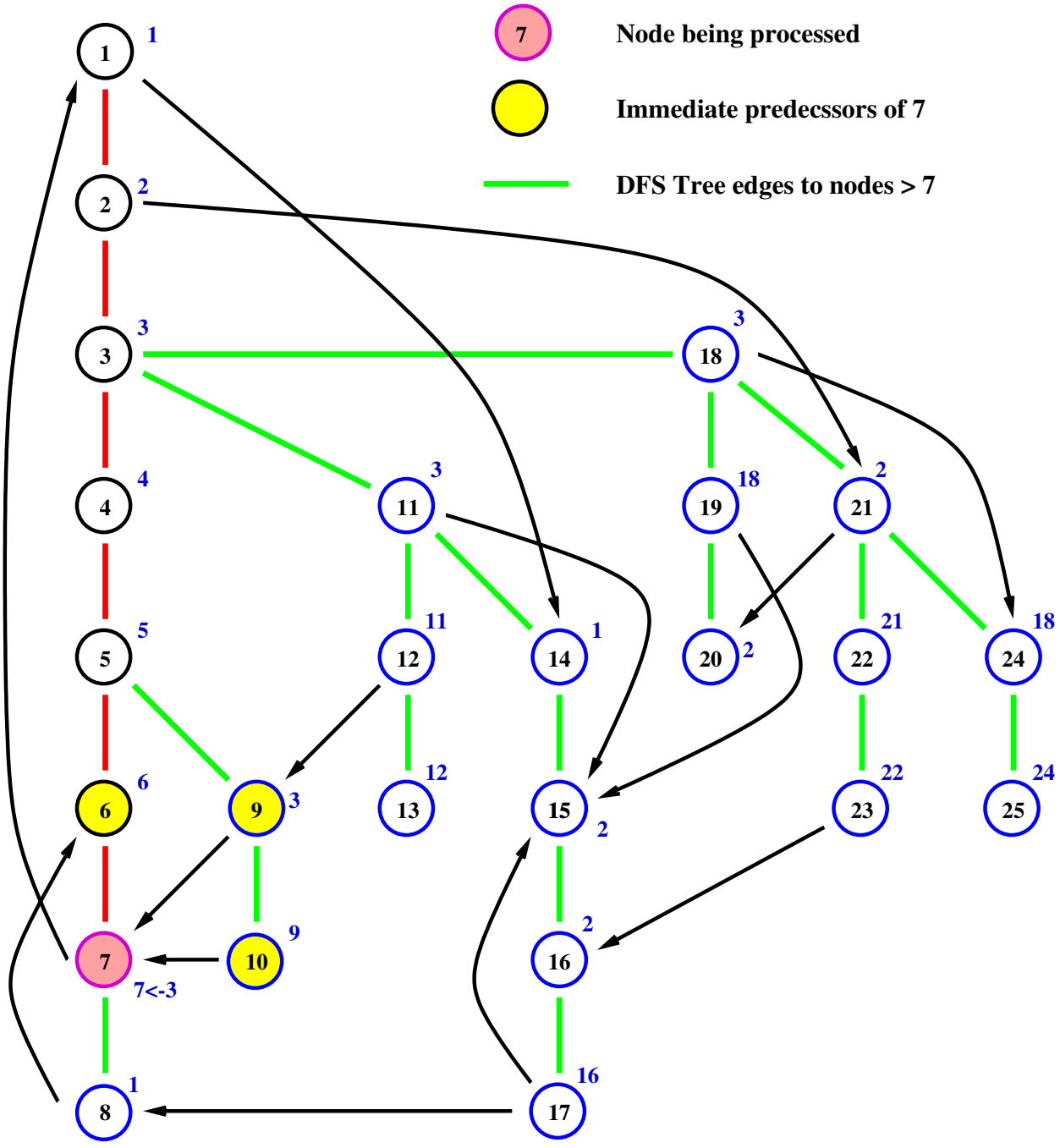
DFS of the Flowgraph



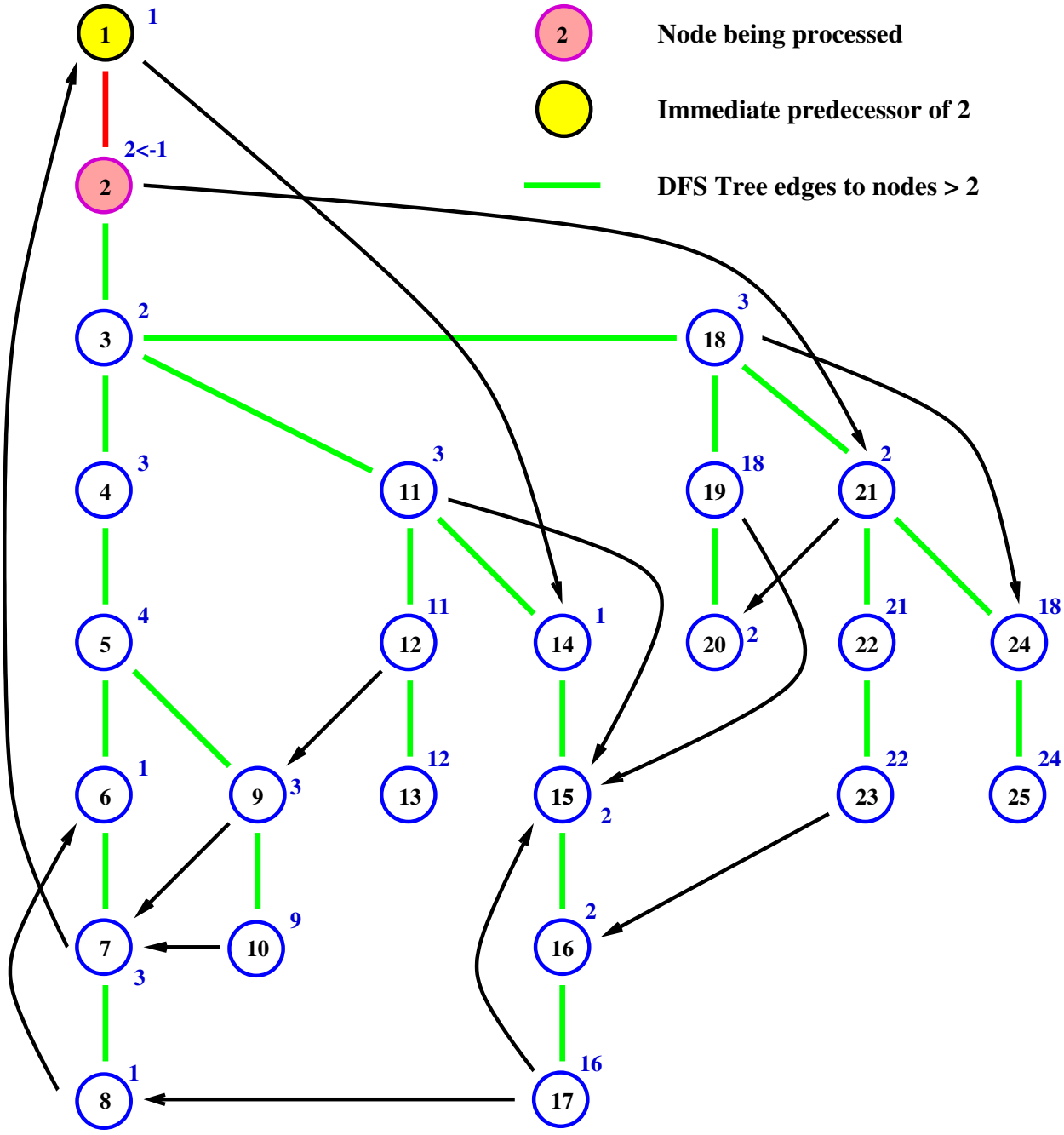
Processing Node 15



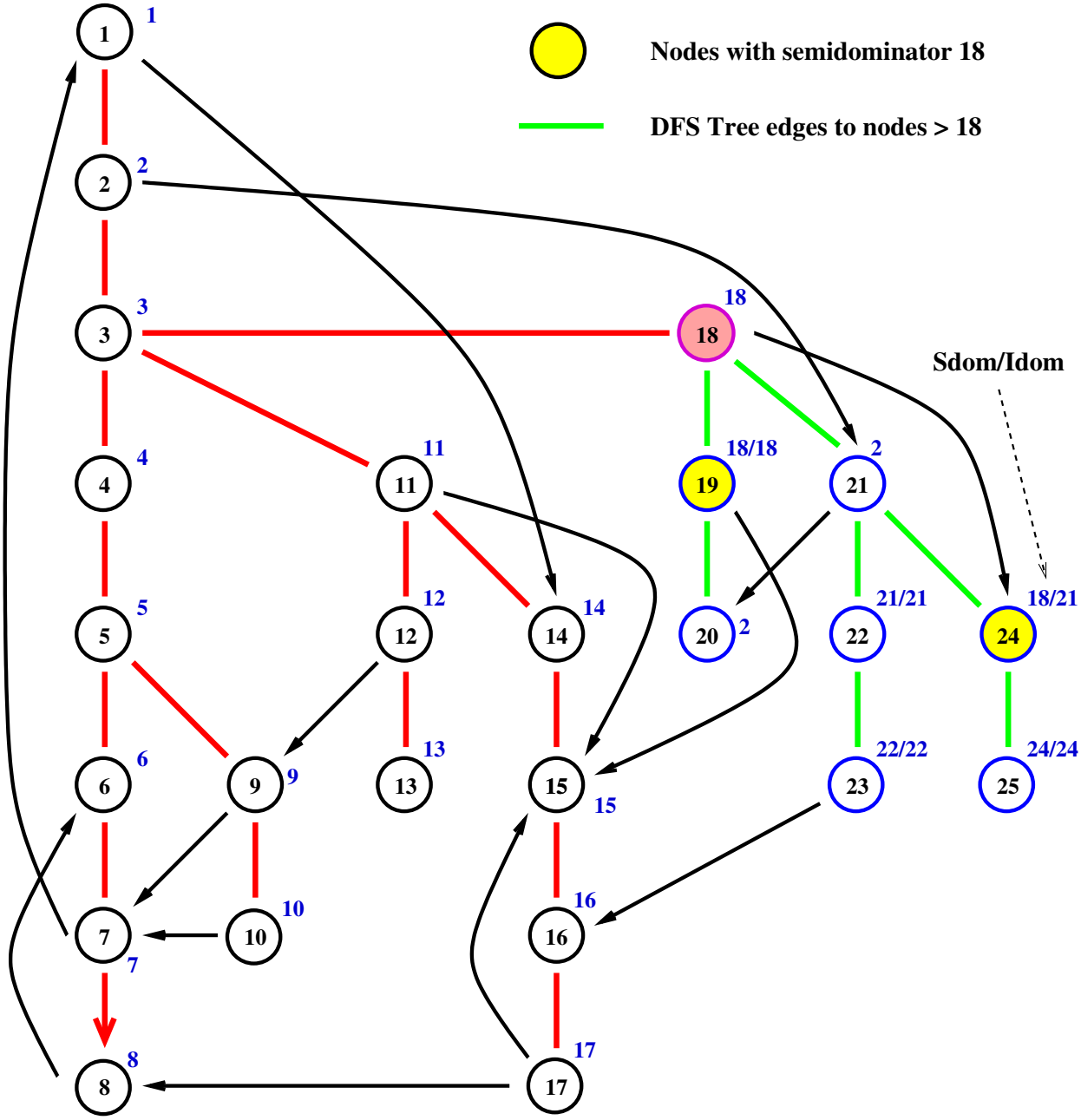
Processing Node 7



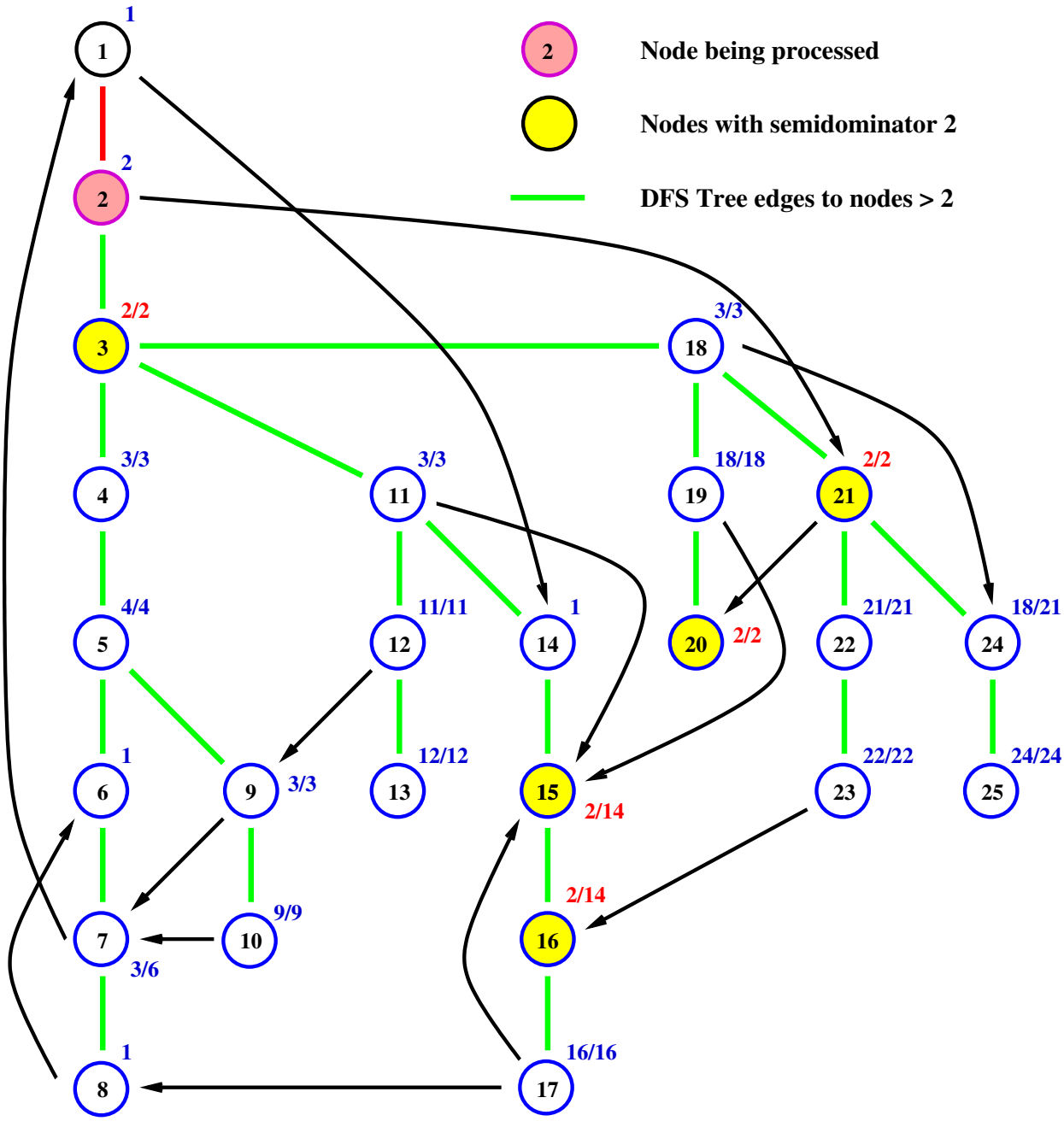
Processing Node 2



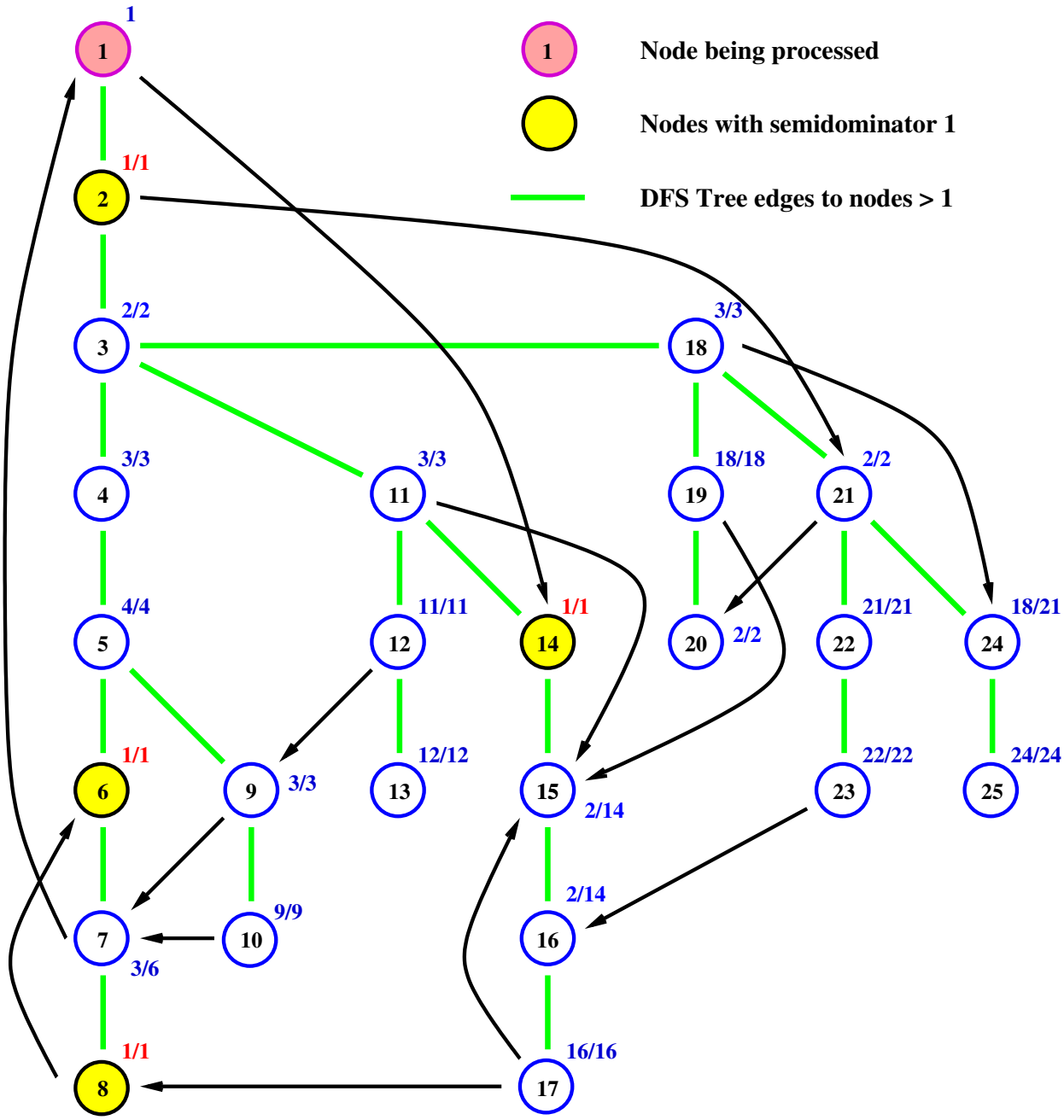
Nodes with Semidominator 18



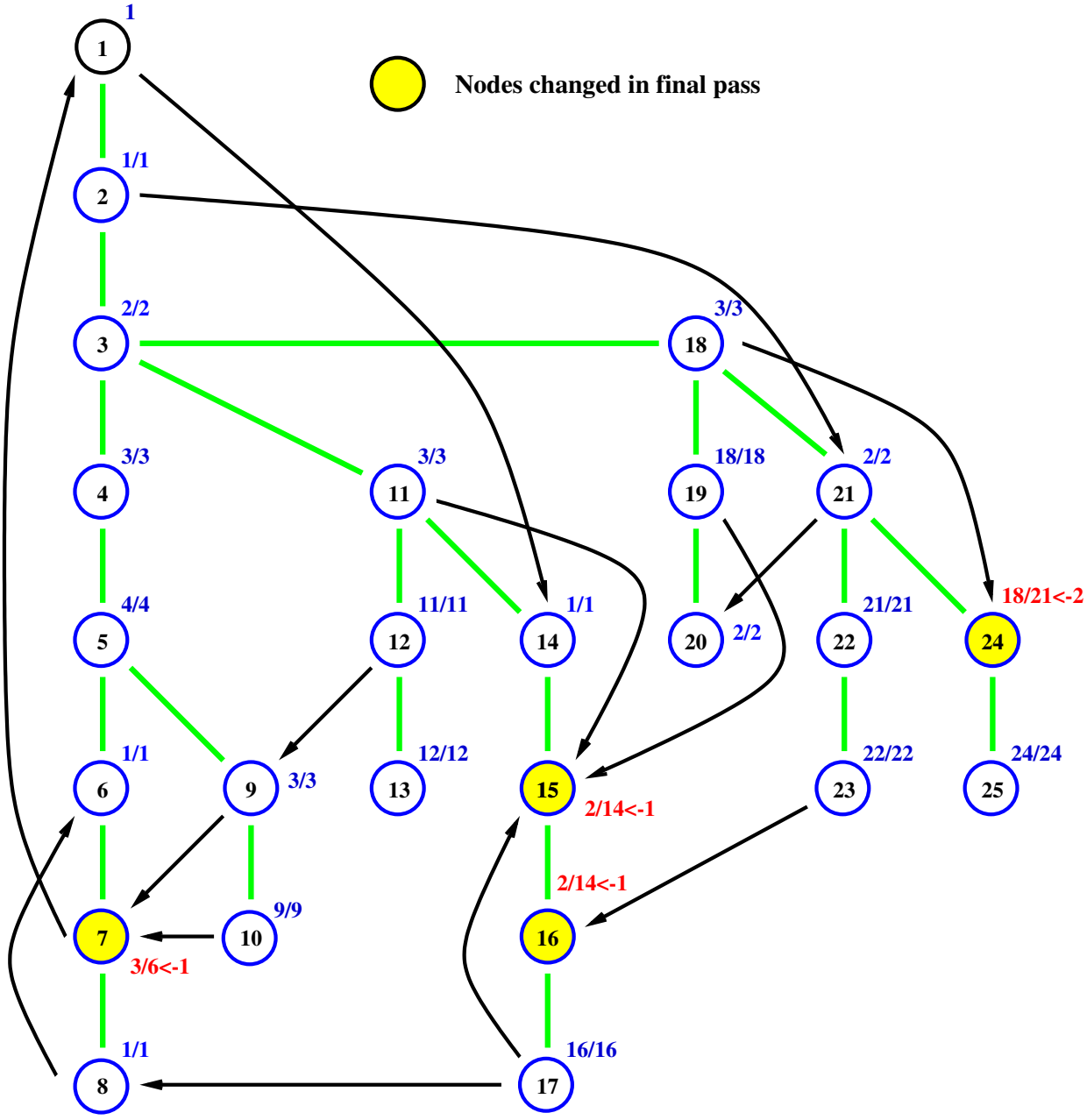
Nodes with Semidominator 2



Nodes with Semidominator 1



Final Phase


Nodes changed in final pass
1
1
1/1
2
2/2
3
3/3
18
3/3
4
3/3
11
18/18
19
2/2
21
4/4
5
11/11
12
1/1
14
2/2
20
21/21
22
18/21<-2
24
1/1
6
3/3
9
12/12
13
2/14<-1
15
22/22
23
24/24
25
2/14<-1
9/9
7
10
3/6<-1
16
1/1
8
16/16
17

Step 1: Initialisation

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

For each vertex v from 1 to `n` set:

```
parent[v]    := DFS tree parent of v
succs[v]     := the given list of successors
preds[v]     := list of predecessors
semi[v]      := v
idom[v]      := 0
ancestor[v]  := 0
best[v]      := v
bucket[v]    := 0
```

Note that indirection in BCPL normally uses expressions such as `parent!v`, but for compatibility with other languages `parent[v]` is also allowed.

Steps 2, 3 and 4

```
FOR w = n TO 2 BY -1 DO
{       LET p = parent[w]

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

step3:  FOR each v in bucket[p]
        { LET u = EVAL(v)
          idom[v] := semi[u]<semi[v] -> u, p
        }
        bucket[p] := 0
}

step4:  FOR w = 2 TO n DO
          UNLESS idom[w] = semi[w] DO
                 idom[w] := idom[idom[w]]
        idom[1] := 0
```

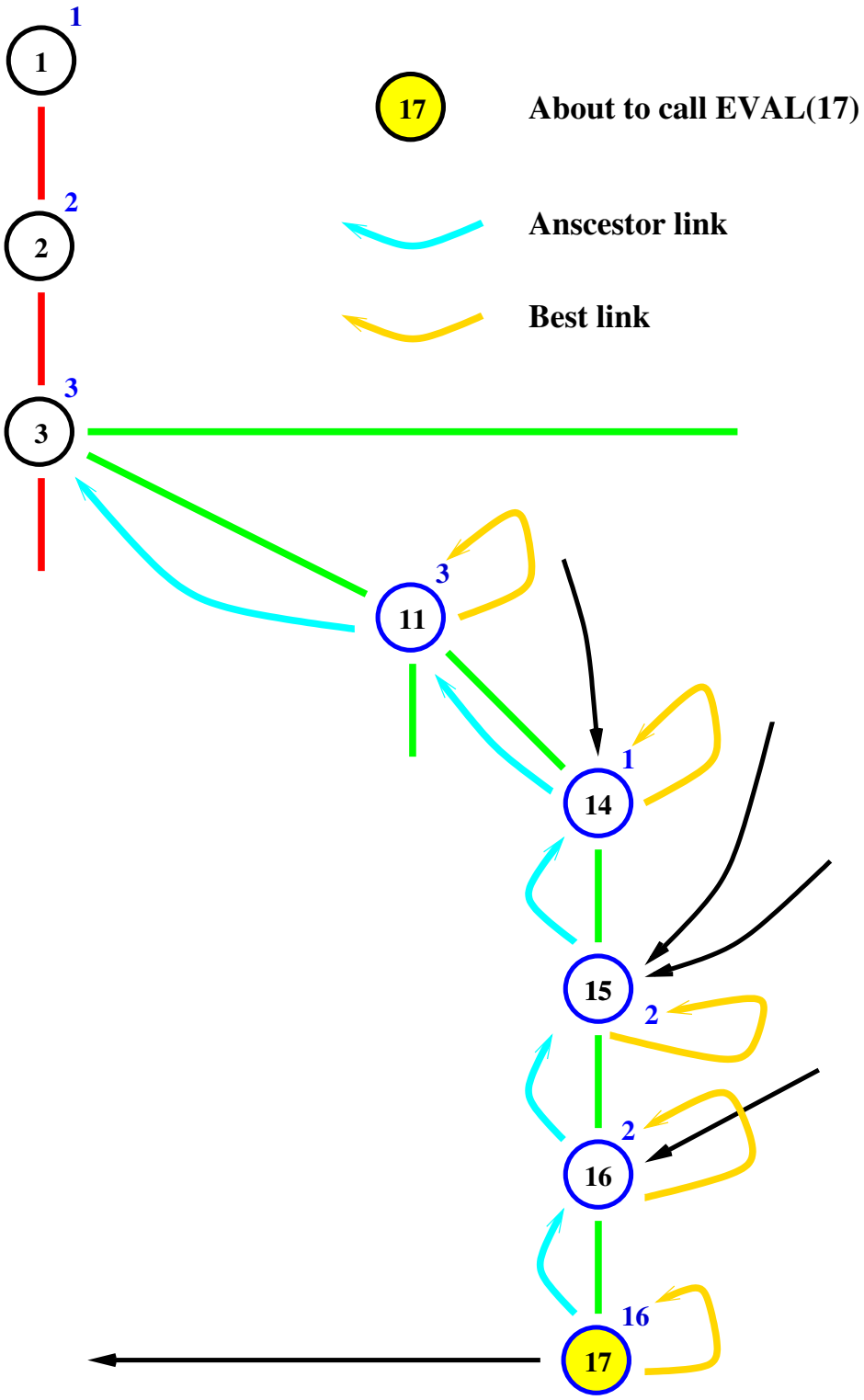
Very Simple LINK and EVAL

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

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

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

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

  RESULTIS v
}
```

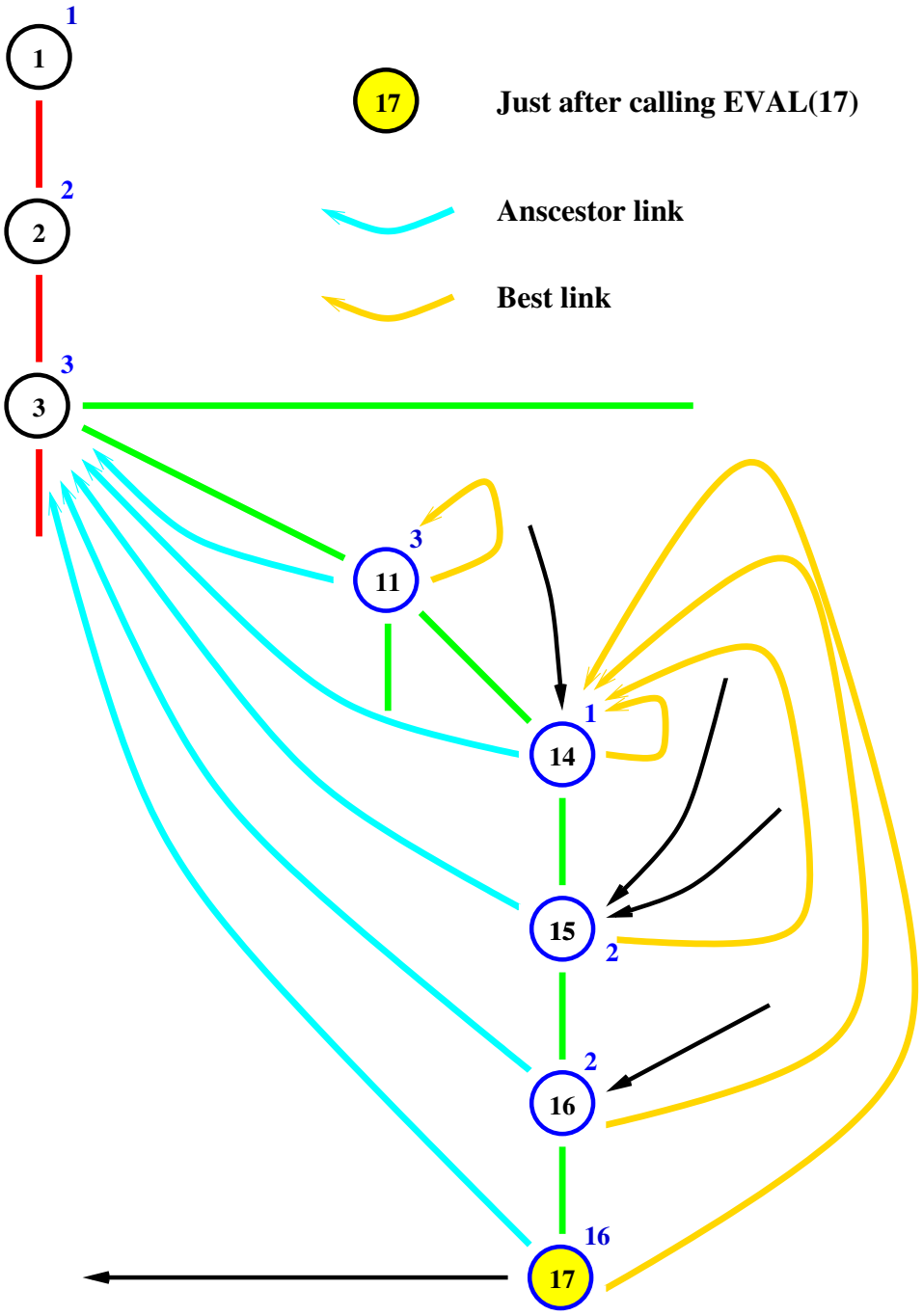
Simple LINK and EVAL

```
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]
}
```

Before calling EVAL(17)



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 wertex in the original chain. This clearly increases the efficiency of later calls of EVAL.

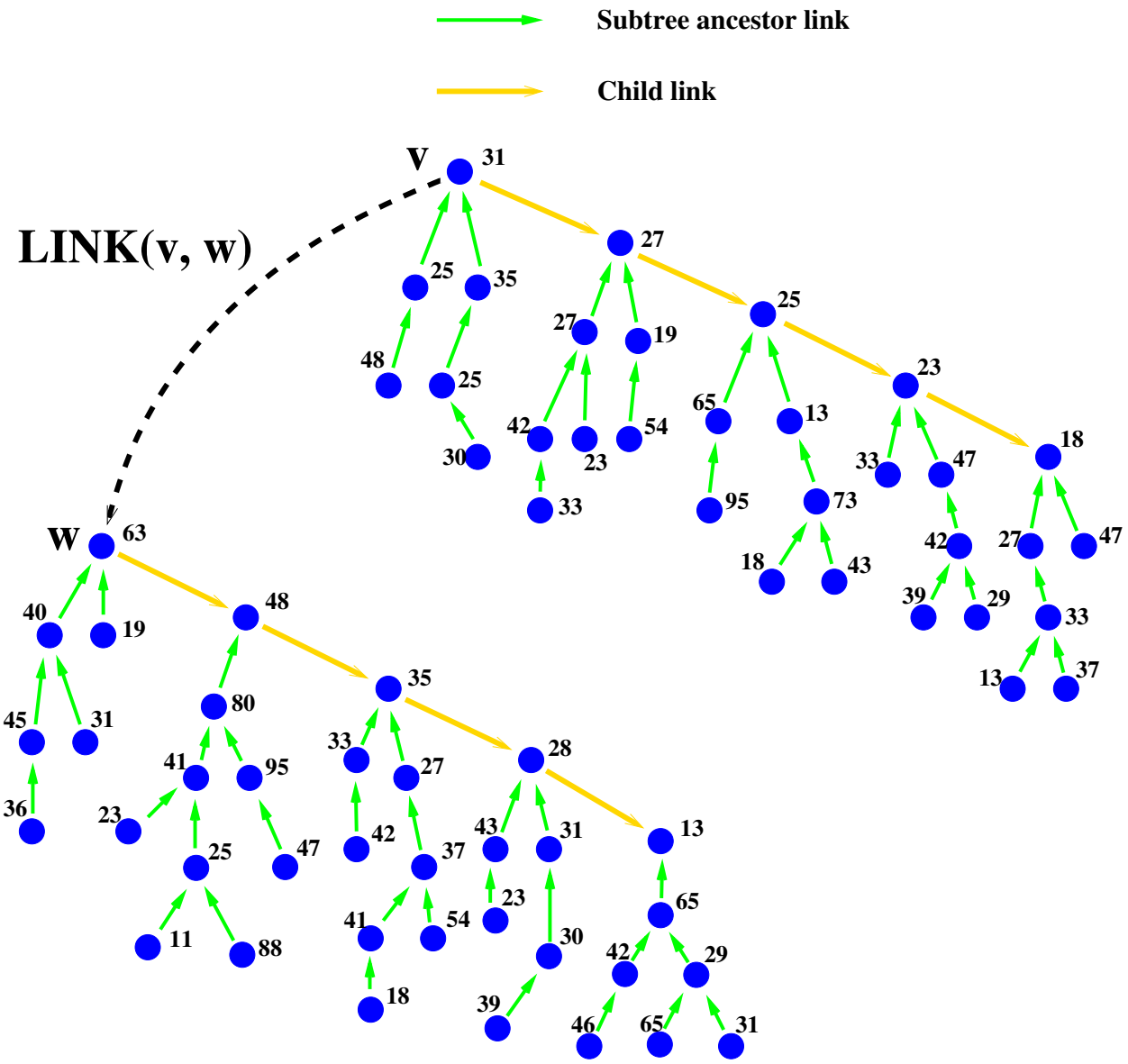
```
LET EVAL(v) = VALOF
{ UNLESS ancestor[v] RESULTIS best[v]
  COMPRESS(v)
  RESULTIS semi[best[ancestor[v]]] < semi[best[v]] ->
                best[ancestor[v]],        best[v]
}
```

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] }
}
```

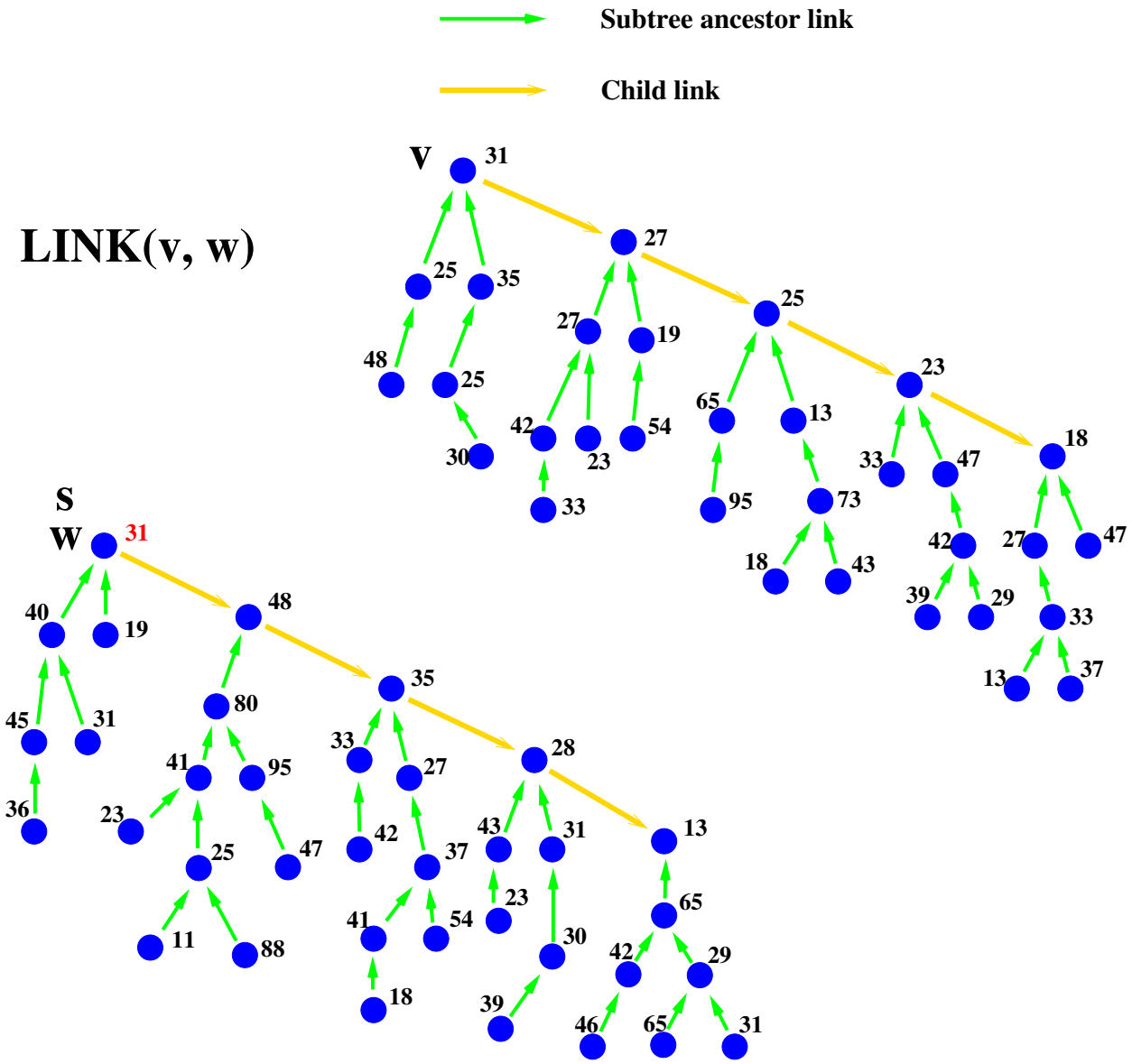
Balanced Trees

The number next to each vertex v in the following diagram is semi[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.

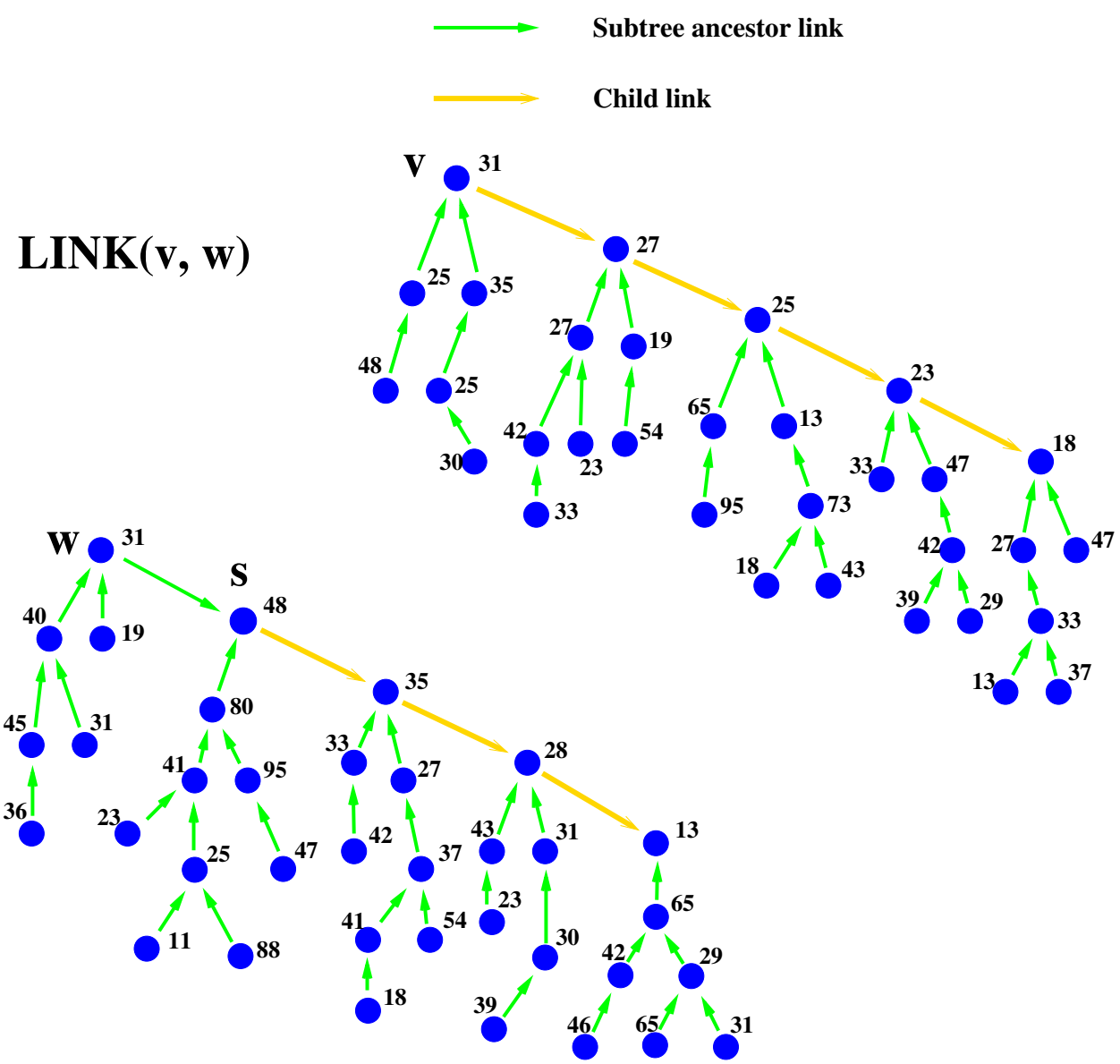


Balanced Trees 1

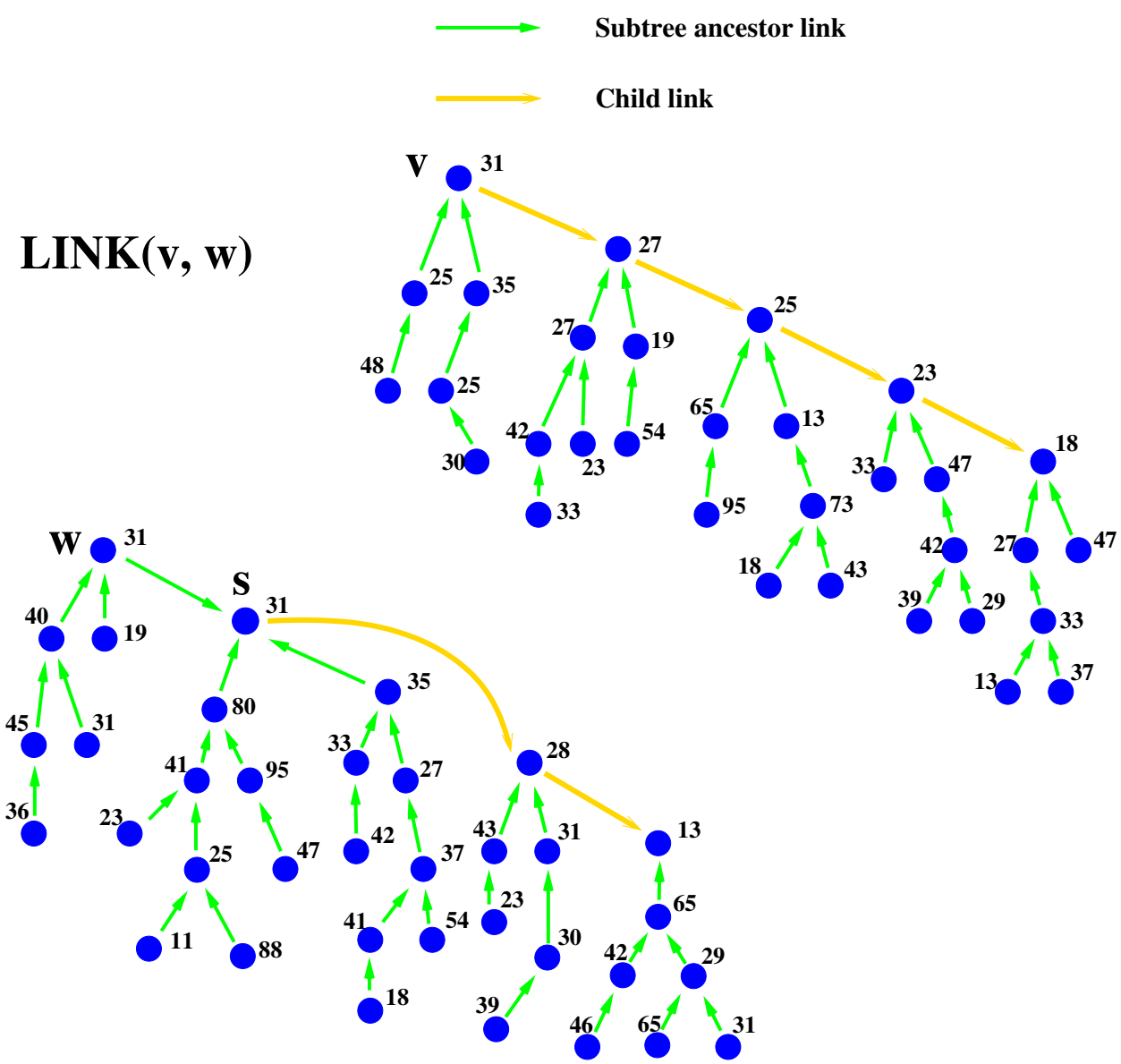
Just before LINK is called the best value of q may have been reduced possibly requiring its child chain to be modified to reinstate its monotonicity.



Balanced Trees 2

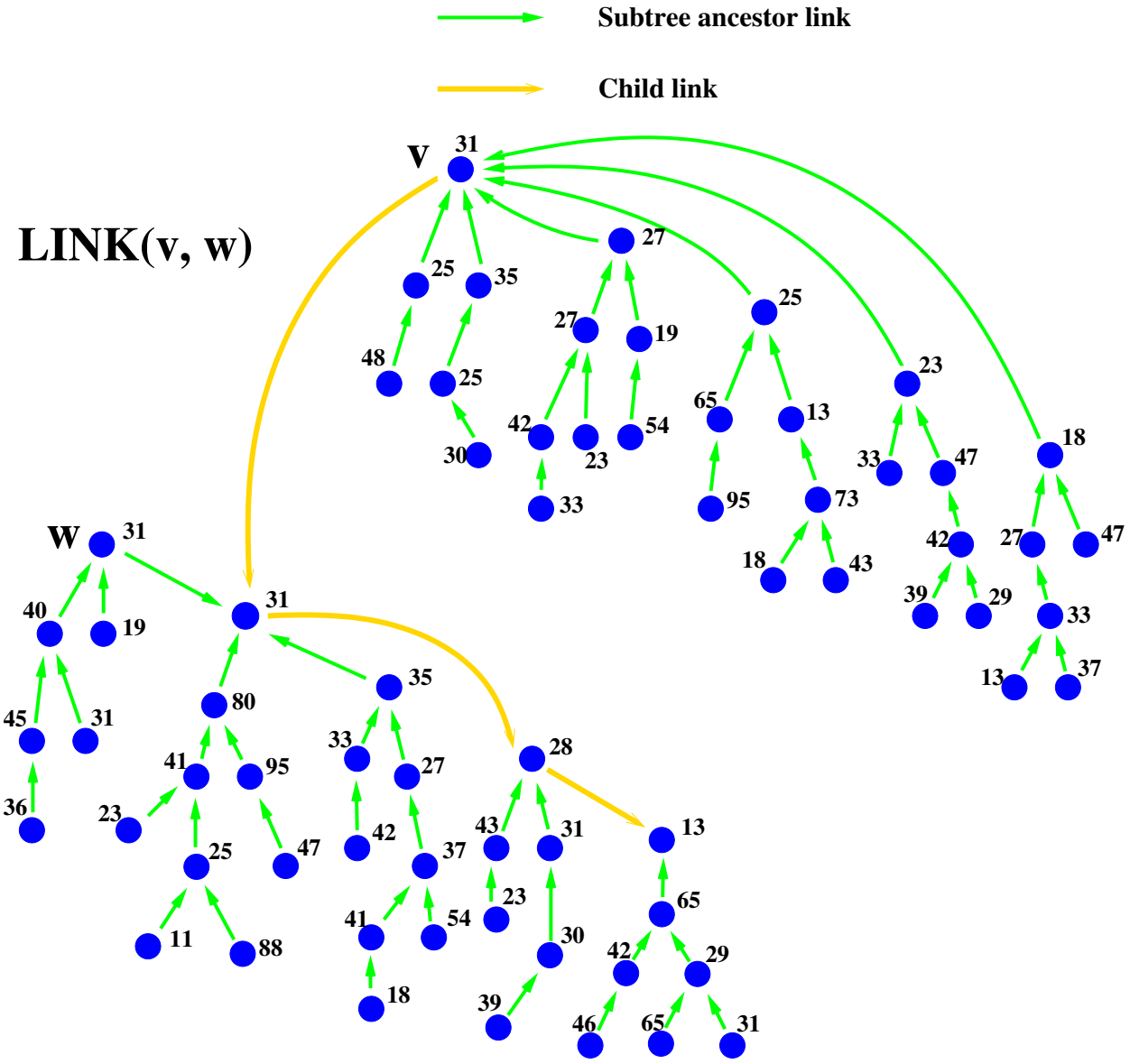


Balanced Trees 3



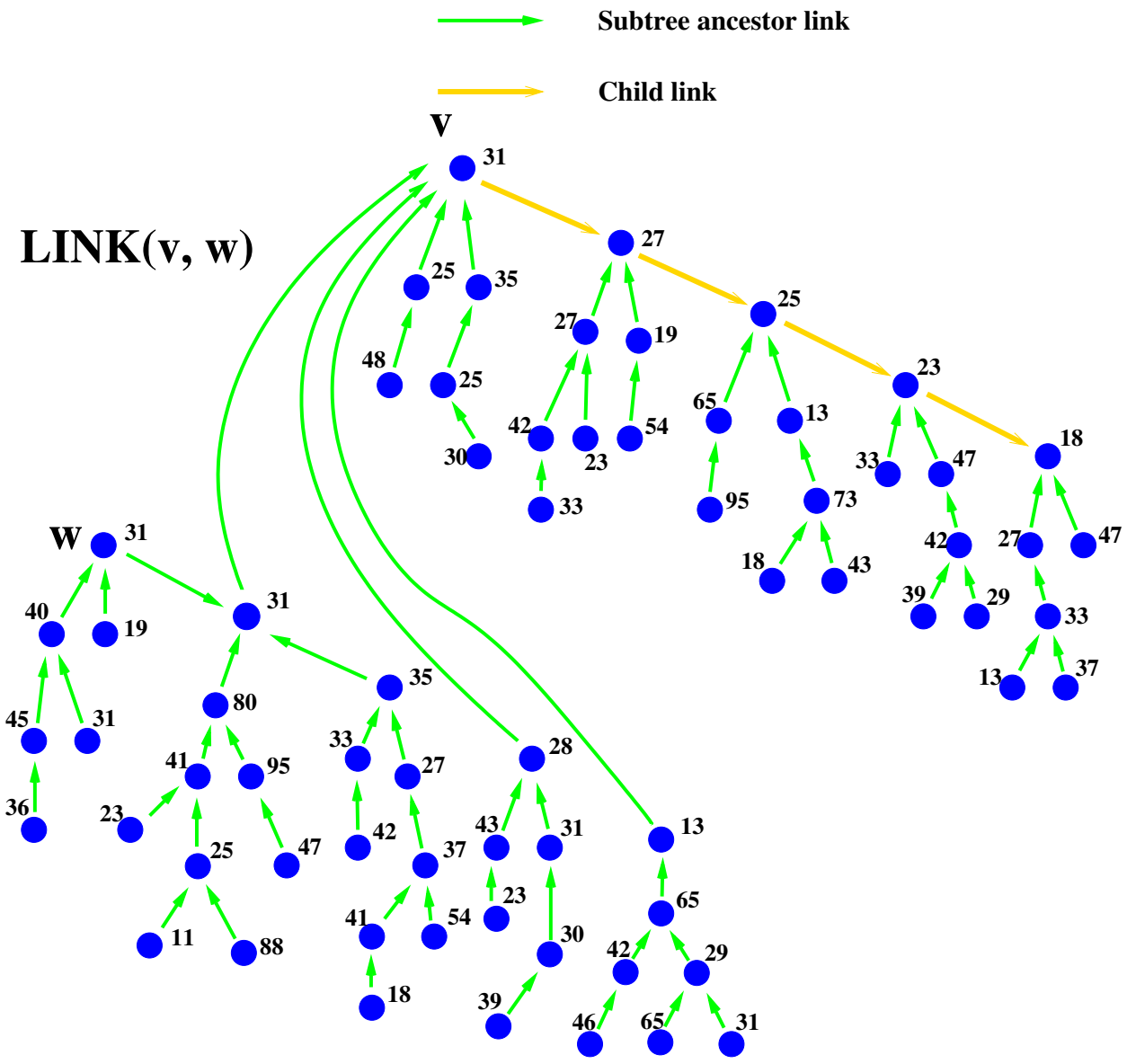
Balanced Trees 4a

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.



Experimental Results

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

Random Graph				Cintcode Instruction Counts		
Nodes	Edges	Seed		v.simple	simple	sophisticated
1000	1500	1		284819	272331	321589
1000	2000	1		455097	317233	358262
1000	2500	1		1180722	376698	388822
1000	3000	1		2440849	445055	416713
1000	5000	1		5680947	630373	542251
1000	10000	1		12848479	1049128	850692
10000	50000	1		334315826	6614589	5432695
10000	100000	1		949583241	10928097	8541841
100000	400000	1		-	56932784	48589754
100000	123289	1	f	24592148	25380561	32042239