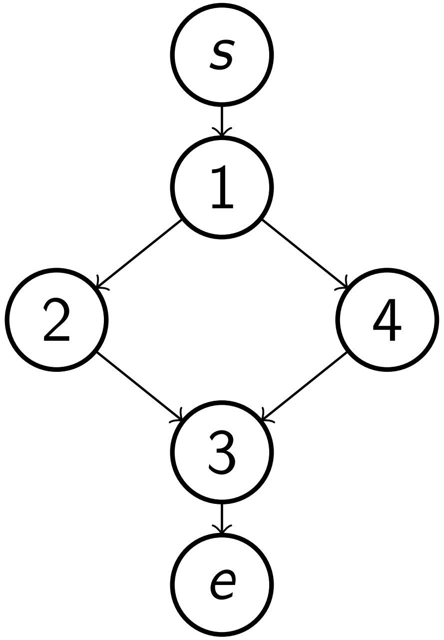
Contents of Lecture 2

- Dominance relation
- An inefficient and simple algorithm to compute dominance
- Immediate dominators
- Dominator tree

Definition of Dominance

- Consider a control flow graph $G(V, E, s, e)$ and two vertices $u, v \in V$.
- If every path from s to v includes u then u **dominates** v.
- For example 1 dominates itself, 2, 3, 4, and e.



Notation and obvious facts

- We write u dominates v as $u \gg v$.
- The set of dominators of a vertex w is written as $dom(w)$, i.e.
- $dom(w) = \{v | v \ggeq w\}$.
- The start vertex has only one dominator: $dom(s) = \{s\}$.
- All vertices are dominated by s.
- If $u \ggeq v$ and $u \neq v$ then we say that u **strictly dominates** v which is written as $u \gg v$.

A restriction on CFG's

- In a CFG, we require that all vertices are on a path from s to e.
- Vertices reachable from s can be detected using depth first search, and then all unvisited vertices can be deleted.
- Due to return statements and infinite loops there can be vertices with no path to e.
- Return-statements are usually collected in one place (in the exit vertex) so a return then is a branch to the exit vertex.
- Infinite loops can be given a "fake" conditional branch (which is always false) in order to create a path to exit.
- In the optimization Dead Code Elimination it's important that every vertex is on a path to e.

Sets and relations

- Assume S and T are sets.
- The Cartesian product $S \times T$ is the set $\{(a, b) | a \in S \wedge b \in T\}$.
- Any subset T of $S \times S$ is a relation on S.
- T is reflexive iff $\forall a \in S, (a, a) \in T$.
- T is irreflexive iff $\forall a \in S, (a, a) \notin T$.
- T is symmetric iff $(a, b) \in T \Rightarrow (b, a) \in T$.
- T is asymmetric iff $(a, b) \in T \Rightarrow (b, a) \notin T$.
- T is antisymmetric iff $(a, b) \in T \wedge (b, a) \in T \Rightarrow a = b$.
- T is transitive iff $(a, b) \in T \wedge (b, c) \in T \Rightarrow (a, c) \in T$.
- A relation which is reflexive, antisymmetric and transitive is called a partial order.
- In a total order such as the integers all elements can be compared but not in a partial order.

Dominance is a partial order

- Dominance is reflexive. Obvious since v must be on any path to itself.
- Dominance is antisymmetric: if both $u \gg\!\!= v$ and $v \gg\!\!= u$ then $u = v$.
  - Assume first that dominance is not antisymmetric and that u and v dominate each other and they are different vertices.
  - Neither u nor v can be s since s is only dominated by itself.
  - Consider a cycle-free path from s to v. It must include u since $u \gg\!\!= v$.
  - But since $v \gg\!\!= u$, we must reach v on that path to u.
  - Now v is twice on the cycle free path which is a contradiction.
  - Hence $u = v$.
- Dominance is transitive: if $u \gg\!\!= v$ and $v \gg\!\!= w$ then $u \gg\!\!= w$
  - Consider any path from s to w.
  - Since $v \gg\!\!= w$, v must be on that path.
  - Since $u \gg\!\!= v$, u must also be on that path.
  - The path was selected arbitrarily which means u is on any such path, i.e. $u \gg\!\!= w$.

Predecessors of a dominated vertex

- If the edge $(v, w) \in E$ of a graph (V, E) then v is a predecessor of w.
- Consider any two vertices $u, v \in V$ and $u \neq v$. Then we have:
- $u \gg v \iff u \gg p_i; \; \forall p_i \in pred(v)$.

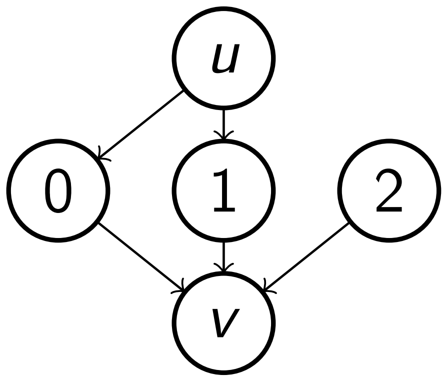
In other words:

- If we want to know if v is dominated by u, we can check if all predecessors of v are dominated by u.
- Then, to find which vertices dominate v, we can check which vertices dominate all predecessors of v, i.e. the intersection of dominators of each predecessor. See below.
- But let us first prove the above statement.

Predecessors of a dominated vertex, continued

Let us consider the $\Rightarrow$ direction first: $u \gg v \;\Rightarrow\; u \gg p_i;\ \forall p_i \in pred(v)$.

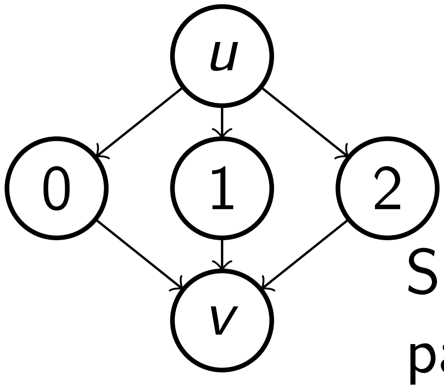
- Assume the contrary, that there exists a predecessor p_i of v which is not dominated by u.
- Then there exists a path $p = (w_0, w_1, w_2, ..., w_k)$ from $s = w_0$ to $p_i = w_k$ which does not include u.
- But then there exists a path $(w_0, w_1, w_2, ..., w_k, w_{k+1})$ from $s = w_0$ to $v = w_{k+1}$ which does not include u, but this is impossible since $u \gg v$.
- Hence, u must dominate every predecessor of v.



if not $u \gg 2$ then it cannot be true that $u \gg v$
u must dominate every predecessor of v to be able to dominate v.

Predecessors of a dominated vertex, continued

Let us then consider the $\Leftarrow$ direction: $u \gg v \;\; \Leftarrow \;\; u \gg p_i;\ \forall p_i \in pred(v)$.

- If u dominates every predecessor of a vertex v then u must also dominate v itself.
- Assume the contrary that there exists a path from s to v which does not include u.
- The second last vertex on that path is a predecessor p_i of v.
- But u dominates every p_i and therefore u must be on the selected path. A contradiction which means $u \gg v$.
- 

Since u dominates every p_i it must be on every path to v and therefore dominate v.

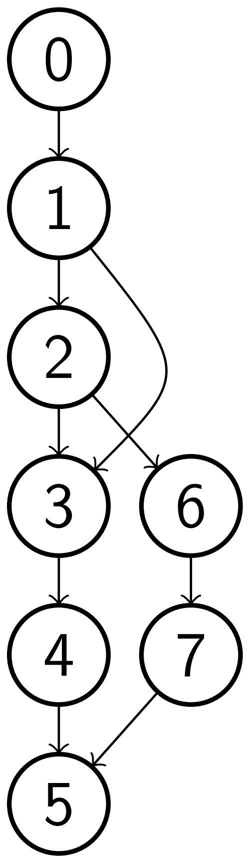
Dominance relation

- Dominance is either computed to say which vertices dominate v,
- or, "what does u dominate" ? (expressed as descendants in a tree)
- We will first look at the first, i.e. computing $dom(v)$
- Recall: $pred(v)$ and $succ(v)$ are sets of *immediate* predecessors and successors, *one* arc from v.

Computing the dominators of each vertex

procedure compute_dominance
    $dom(s) \leftarrow \{s\}$
    for each $w \in V - \{s\}$ **do**
        $dom(w) \leftarrow V$
    $change \leftarrow true$
    while $change$ **do**
        $change \leftarrow false$
        for each $w \in V - \{s\}$ **do**
            $old \leftarrow dom(w)$
            $dom(w) \leftarrow \{w\} \cup \bigcap_{p \in pred(w)} dom(p)$
            if $old \neq dom(w)$
                $change \leftarrow true$
end

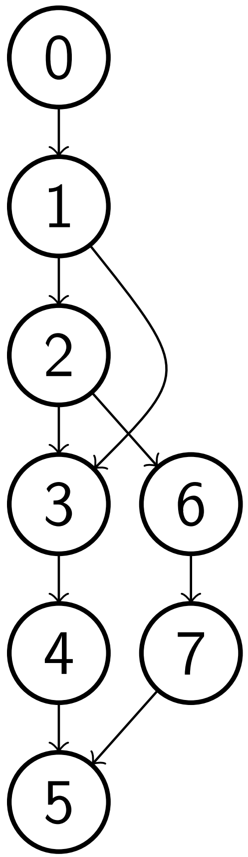
An Example Control Flow Graph 1(3)



$$dom(w) \leftarrow \{w\} \cup \bigcap_{p \in pred(w)} dom(p)$$

vertex	init.	1st iter.
0	$\{0\}$	$\{0\} = \{ 0 \}$
1	V	$\{1\} \cup \{0\} = \{0, 1\}$
2	V	$\{2\} \cup \{0, 1\} = \{0, 1, 2\}$
3	V	$\{3\} \cup (\{0, 1, 2\} \cap \{0, 1\}) = \{0, 1, 3\}$
4	V	$\{4\} \cup \{0, 1, 3\} = \{0, 1, 3, 4\}$
5	V	$\{5\} \cup (\{0, 1, 3, 4\} \cap V) = \{0, 1, 3, 4, 5\}$
6	V	$\{6\} \cup \{0, 1, 2\} = \{0, 1, 2, 6\}$
7	V	$\{7\} \cup \{0, 1, 2, 6\} = \{0, 1, 2, 6, 7\}$

An Example Control Flow Graph 2(3)

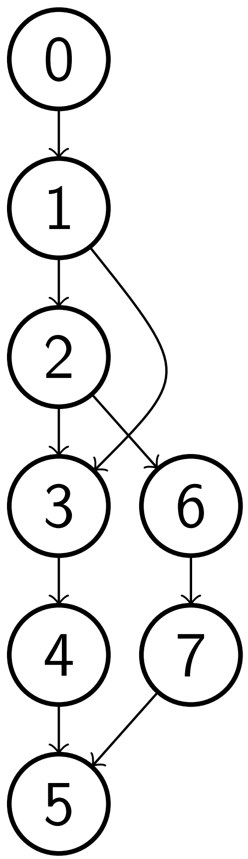


$$dom(w) \leftarrow \{w\} \cup \bigcap_{p \in pred(w)} dom(p)$$

vertex	1st iter.	2nd iter.
0	$\{0\}$	same
1	$\{0, 1\}$	same
2	$\{0, 1, 2\}$	same
3	$\{0, 1, 3\}$	same
4	$\{0, 1, 3, 4\}$	same
5	$\{0, 1, 3, 4, 5\}$	$\{5\} \cup (\{0, 1, 3, 4\} \cap \{0, 1, 2, 6, 7\})$
6	$\{0, 1, 2, 6\}$	same
7	$\{0, 1, 2, 6, 7\}$	same

After the third iteration also $dom(5) = \{0, 1, 5\}$ will remain the same and the algorithm terminates.

An Example Control Flow Graph 3(3)



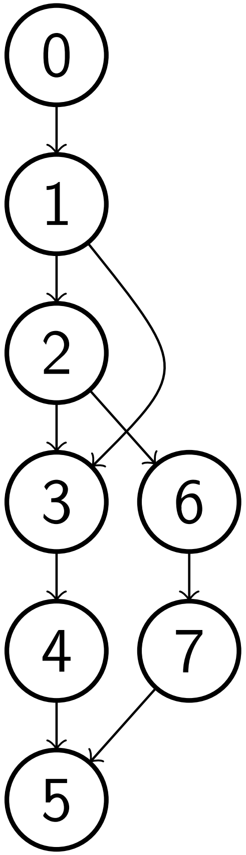
$$dom(w) \leftarrow \{w\} \cup \bigcap_{p \in pred(w)} dom(p)$$

vertex	3rd iter. $dom(w)$
0	$\{0\}$
1	$\{0, 1\}$
2	$\{0, 1, 2\}$
3	$\{0, 1, 3\}$
4	$\{0, 1, 3, 4\}$
5	$\{0, 1, 5\}$
6	$\{0, 1, 2, 6\}$
7	$\{0, 1, 2, 6, 7\}$

Immediate dominators

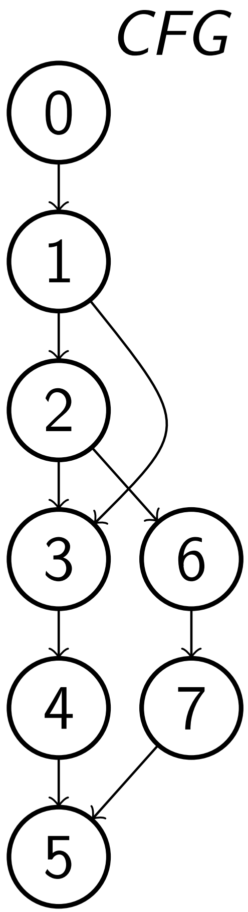
- The set $dom(w)$ is a total order.
- In other words: if $u, v \in dom(w)$ then either $u \gtrgtr v$ or $v \gtrgtr u$.
- We can order all vertices in $dom(w)$ to find the "closest" dominator of w.
- First let $S \leftarrow dom(w) - \{w\}$.
- Consider any two vertices in S.
- Remove from S the one which dominates the other. Repeat.
- The only remaining vertex in S is the **immediate dominator** of w.
- We write the immediate dominator of w as $idom(w)$.
- Every vertex, except s, has a unique immediate dominator.
- We can draw the immediate dominators in a tree called the **dominator tree**, abbreviated **DT**.

The Dominator Tree of Example CFG 1(3)

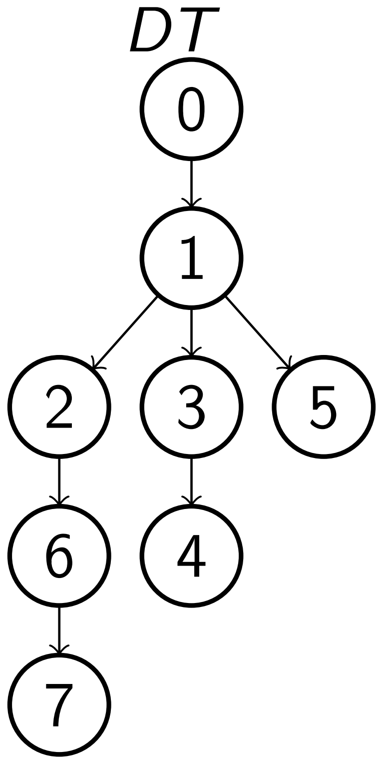


vertex	$dom(w) - \{w\}$	$idom(w)$	how to find idom
0	$\emptyset$	-	has no idom
1	$\{0\}$	0	only 0
2	$\{0, 1\}$	1	remove 0
3	$\{0, 1\}$	1	remove 0
4	$\{0, 1, 3\}$	3	remove 0,1
5	$\{0, 1\}$	1	remove 0
6	$\{0, 1, 2\}$	2	remove 0,1
7	$\{0, 1, 2, 6\}$	6	remove 0,1,2

The Dominator Tree of Example CFG 2(3)

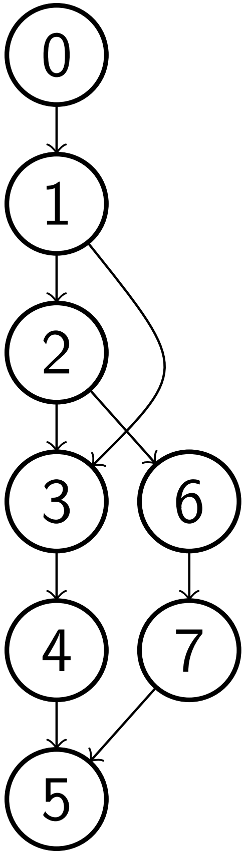


w	$idom(w)$
0	-
1	0
2	1
3	1
4	3
5	1
6	2
7	6

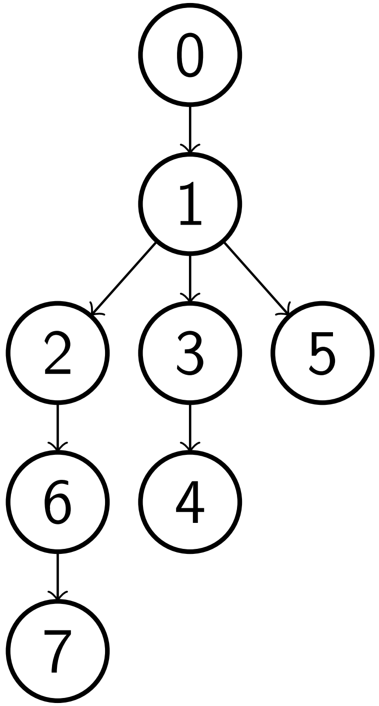


The Dominator Tree of Example CFG 3(3)

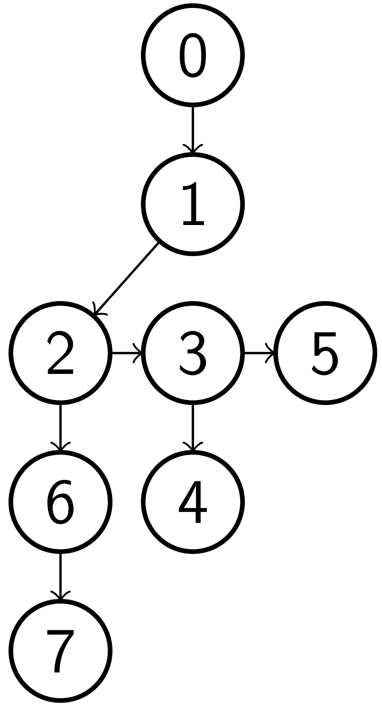
CFG



DT



domchild and domsibling



The children of a vertex in the DT are a set (and not ordered).

How to construct the dominator tree

- Assume we know the *idom*(w) of each vertex (except s).
- How should we construct the DT?
-
```
typedef struct vertex_t vertex_t;
struct vertex_t {
        vertex_t*       idom;
        vertex_t*       domchild;
        vertex_t*       domsibling;
};
```
- Of course both `domchild` and `domsibling` initially are null pointers.
- Suppose you have just computed *idom*(w) and have a pointer to w.
- How do you link it into the DT without using any conditional branch instruction?

Link *w* into *DT*

- Don't check for the case of `domchild` or `domsibling` being a null pointer...

```
w->domsibling = w->idom->domchild;
w->idom->domchild = w;
```

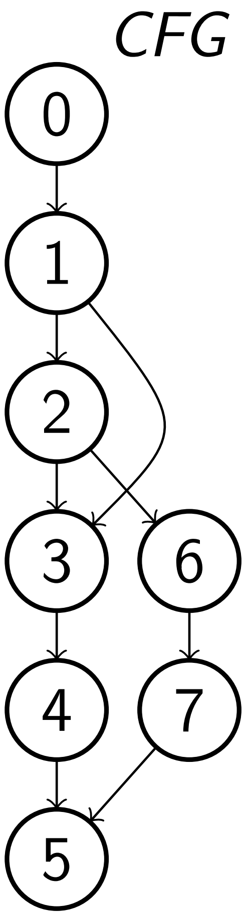
Summary so far

- The iterative algorithm we saw is an example of **iterative dataflow analysis**.
- Dataflow analysis concerns the flow of values but the technique is identical to what we saw.
- The sets are represented as bit-vectors.
- Usually about three iterations suffice.
- It doesn't matter for correctness in which order we inspect the vertices in each iteration but to improve the speed of the compiler, there are preferences (see below).
- We will see an algorithm which is faster and constructs the dominator tree directly.
- Given the set $dom(w)$ it takes (as we saw) additional effort to construct the dominator tree.

In which order should we process the vertices?

- The information flows forward so it is better to have processed the predecessors of a vertex w before w itself is processed.
- We put each vertex in an array in **reverse post order**.

Reverse post order



- An array is allocated to hold each vertex.
- The array will be processed with increasing indexes.
- The vertices are put into the array starting at the highest index.
- The last vertex put into the array is s at index 0.
- Do a depth first search as follows
- When a vertex has no unvisited successor, put it at the last free position in the array.
-

0	1	2	6	7	3	4	5

- This way we will have processed both 4 and 7 before computing $dom(5)$.

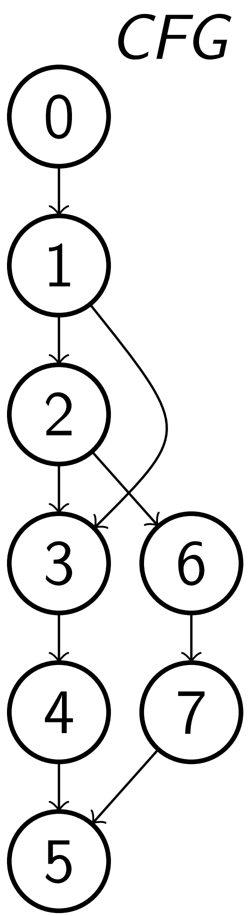
Computing idom and DT faster

- The LT algorithm was completed in 1979 by Robert Tarjan and his PhD student Thomas Lengauer at Stanford.
- Thomas Lengauer is the brother of Christian Lengauer whose group in Passau has developed many high order transformations (and visited Lund in 1992).
- The LT algorithm calculates the immediate dominator and is based on insights from depth first search.
- We will focus on understanding the key ideas of the algorithm.

The Semi-Dominator of a Vertex

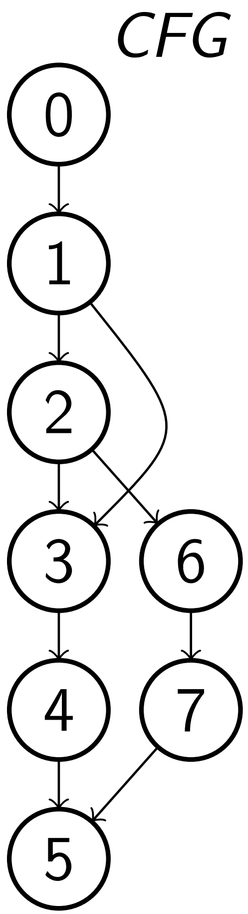
- The semi-dominator of a vertex is much easier to compute than the immediate dominator and is almost always identical to the immediate dominator.
- We will soon define the semi-dominator.
- The idea is to find the semi-dominator which is easy, and then determine whether the semi-dominator also is the immediate dominator.
- If it's not, then the immediate dominator of w is the immediate dominator of a certain ancestor between w and $sdom(w)$ in the DFS tree (explained below).

Definition of the Semi-Dominator of a Vertex



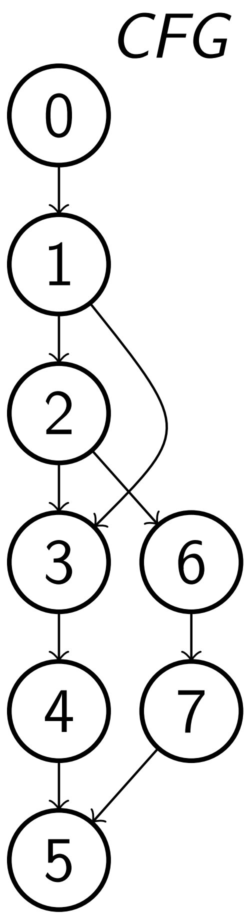
- First a depth first search numbering is performed on the CFG. This is shown to the left.
- When we write $u < v$ we mean that u has a lower depth first search number than v.
- The semi-dominator of a vertex w is the smallest vertex v such that there is a path $(v_0, v_1, v_2, ..., v_k)$ from $v = v_0$ to $w = v_k$ with $v_i > w$ for $1 \leq i \leq k - 1$, and is written $sdom(w)$.
- For example $sdom(5) = 2$ since the path $(2, 6, 7, 5)$ starts with 2 which is lower than 4 in the alternative path $(4, 5)$.
- Please start with the left most edge during DFS search on the exam!

More about Semi-Dominators and Immediate Dominators



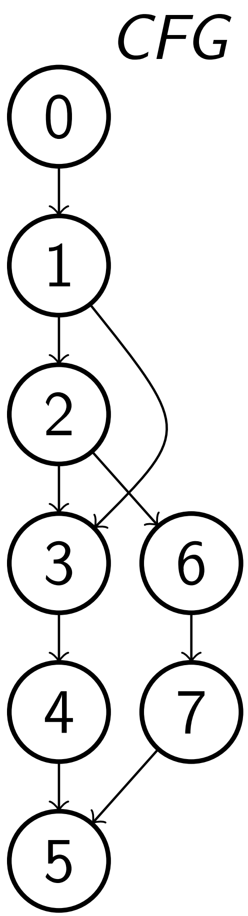
- Consider again vertex 5. We have $sdom(5) = 2$ and $idom(5) = 1$.
- Assume we know how to compute the semi-dominators — it's not very difficult — we only have to find a suitable path.
- What is the "problem" which is the root cause that makes the semi-dominator can be different from the immediate dominator?
- Answer: there is an edge from a vertex coming in from "outside" and between the vertex 5 and the semi-dominator, i.e. the edge $(1, 3)$.
- This is the key problem the algorithm has to deal with.
- Let us next find a way to compute the semi-dominators.

Computing the Semi-Dominators



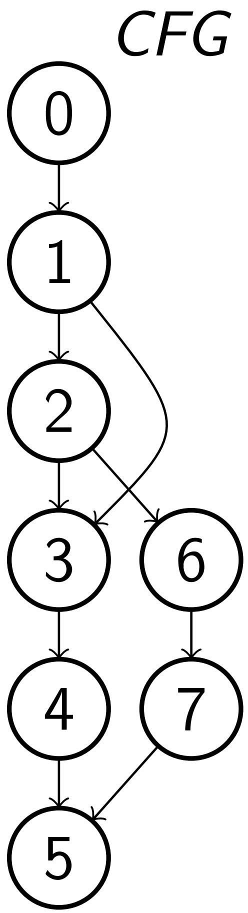
- Recall: the semi-dominator of a vertex w is the smallest vertex v such that there is a path $(v_0, v_1, v_2, ..., v_k)$ from $v = v_0$ to $w = w_k$ with $v_i > w$ for $1 \leq i \leq k-1$, and is written $sdom(w)$.
- We can see there can be multiple candidates for being the semi-dominator.
- Any path to w obviously must end with an edge to w from a predecessor of w.
- All predecessors of w are searched for a possible candidate path and semi-dominator.
- Note that the path may consist of only one edge.
- How far should we search backwards???

Computing the Semi-Dominators



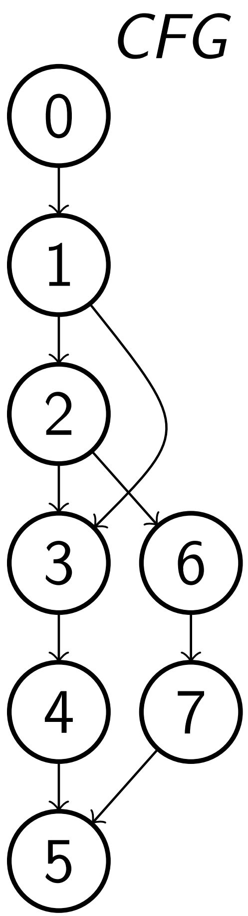
- How far should we search backwards???
- Recall we want to find a path $(v = w_0, w_1, w_2, ...w_{k-1}, w_k = w)$ where $w_i > w$ for $1 \leq i < k$.
- Therefore we should only search backwards on vertices with a higher number than w.
- This is achieved as follows: the Lengauer-Tarjan algorithm first processes each vertex in **decreasing** depth-first search number.
- We may only search backwards from one vertex to its ancestor in the depth first search tree.
- The function to find a semi-dominator candidate is called **eval** and it finds the ancestor with the least semi-dominator.

Link and Eval



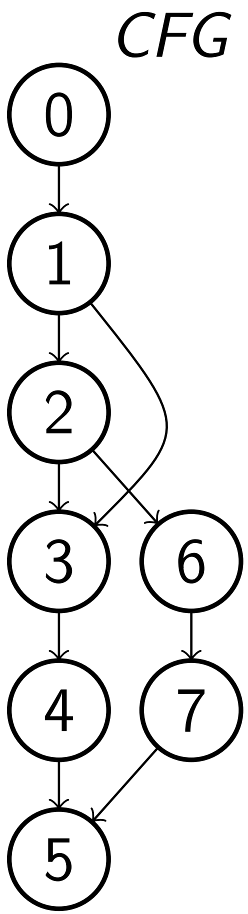
- To limit the search backwards (or actually upwards in the depth first search tree) a separate attribute identical to the parent in the depth first search tree is maintained.
- When a vertex w has been processed, its attribute `w->parent` is copied to `w->ancestor` by the function **link**.
- The function eval uses the `w->ancestor` to search upwards in the depth first search tree.
- The ancestor with least semi-dominator number is returned from eval.
- For all predecessors p_i of w, the smallest return value from $eval(p_i)$ is the semi-dominator of w.

Sdom and Idom



- To determine whether the semi-dominator is the immediate dominator, a search from w to $sdom(w)$ is performed following the `w->ancestor` attributes.
- First of all, the $sdom(w)$ must be an ancestor of w in the DFS tree.
- If any ancestor v in that search has $sdom(v)$ which is lower than $sdom(w)$ then there is an edge which makes it impossible for $sdom(w)$ to be $idom(w)$.
- Therefore, a vertex w is put in a set, called the bucket, in $sdom(w)$.
- The bucket is emptied when a child of $sdom(w)$ is processed.
- When the bucket is emptied, the search from each w in the bucket towards $sdom(w)$ is performed.

Link and Eval



- In the search mentioned on the previous slide, if no ancestor with a lower semi-dominator was found, then we know that $idom(w) = sdom(w)$.
- Otherwise, let u be the ancestor with least semi-dominator found in the search.
- It turns out that $idom(w) = idom(u)$;
- But we don't yet know $idom(u)$ and therefore must record u as an attribute of w.
- It's put in the attribute `w->idom`.
- After all vertices have been processed and found their *sdom*, the vertices are processed again with increasing DFS number to determine the immediate dominator unless already known.

Summary of notation

G	Control flow graph CFG .
T	A depth-first spanning tree of G.
DT	The dominator tree of G.
w	The depth-first search number of vertex w in T.
$v < w$	v has a lower depth-first search number than w.
$v \xrightarrow{*} w$	v is an ancestor of w in T.
$v \xrightarrow{+} w$	v is a proper ancestor of w: $v \xrightarrow{*} w$ and $v \neq w$.
$parent(w)$	parent of w in T.
$ancestor(w)$	also parent of w in T.

The Lengauer-Tarjan Algorithm 1(6)

int *df* /* Depth-first search number. */

procedure $dfs(v, vertex[])$
    $dfnum(v) \leftarrow df$
    $vertex[df] \leftarrow v$
    $sdom(v) \leftarrow v$
    $ancestor(v) \leftarrow null$
    $df \leftarrow df + 1$

    for each $w \in succ(v)$ **do**
        if $(sdom(w) = null)$ {
            $parent(w) \leftarrow v$
            $dfs(w)$
        }

function *eval*(v)
    vertex u

    /* Find ancestor with least sdom. */
    $u \leftarrow v$
    while ($ancestor(v) \neq nil$) **do**
        if ($dfnum(sdom(v)) < dfnum(sdom(u))$)
            $u \leftarrow v$
        $v \leftarrow ancestor(v)$
    return u

procedure *link*(v, w)
    $ancestor(w) \leftarrow v$

The Lengauer-Tarjan Algorithm 3(6)

```
procedure dominators(V, s)
    int    i
    int    n = |V|
    vertex vertex[n]

    /* Step 1. */
    for each w ∈ V do
        sdom(w) ← nil
        bucket(w) ← ∅

    df ← 0
    dfs(s)
```

The Lengauer-Tarjan Algorithm 4(6)

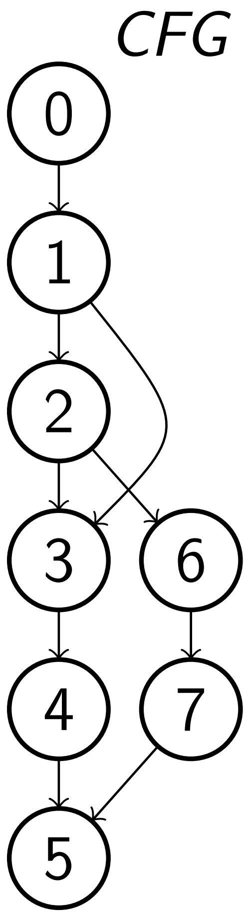
```
for (i ← n − 1; i > 0; i ← i − 1) do {
    /* Step 2. */
    w ← vertex[i]
    for each v ∈ pred(w) do {
        u ← eval (v)
        if (dfnum(sdom(u)) < dfnum(sdom(w)))
            sdom(w) ← sdom(u)
    }
    add w to bucket(sdom(w))

    link(parent(w), w)
```

The Lengauer-Tarjan Algorithm 5(6)

```
/* Step 3. */
for each v ∈ bucket(parent(w)) do {
    remove v from bucket(parent(w))
    u ← eval (v)
    if (dfnum(sdom(u)) < dfnum(sdom(v)))
        idom(v) ← u
    else
        idom(v) ← parent(w)
}
}
```

The Lengauer-Tarjan Algorithm 6(6)

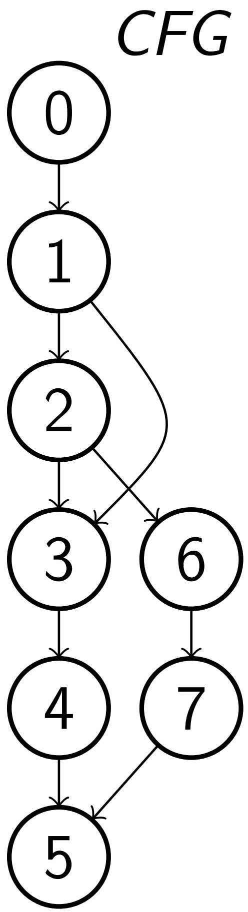
/* Step 4. */
for $(i \leftarrow 1;\ i < n;\ i \leftarrow i + 1)$ {
    $w \leftarrow vertex[i]$
    if $(idom(w) \neq sdom(w))$
        $idom(w) \leftarrow idom(idom(w))$
}
$idom(s) \leftarrow -1$
}



- After Initialization in Step 1.
- $sdom(w) = w$

vertex	parent	bucket	ancestor	sdom	idom
0	-	∅	-	0	-
1	0	∅	-	1	-
2	1	∅	-	2	-
3	2	∅	-	3	-
4	3	∅	-	4	-
5	4	∅	-	5	-
6	2	∅	-	6	-
7	6	∅	-	7	-

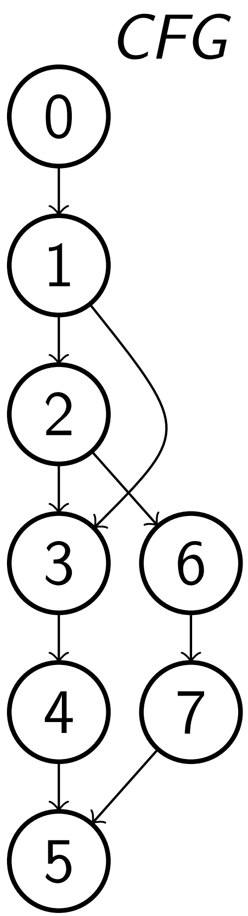
Processing Vertex 7: Step 2



- The only predecessor of $w = 7$ is $v = 6$ which evaluates to $u = 6$.
- $sdom(w = 7)$ becomes 6, and 7 is added to the bucket of its sdom.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\emptyset$	-	1	-
2	1	$\emptyset$	-	2	-
3	2	$\emptyset$	-	3	-
4	3	$\emptyset$	-	4	-
5	4	$\emptyset$	-	5	-
6	2	$\{7\}$	-	6	-
7	6	$\emptyset$	6	6	-

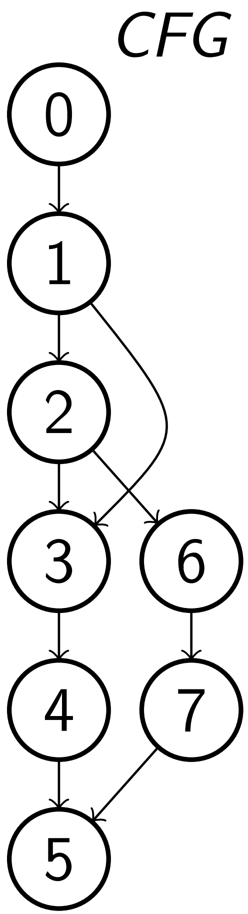
Processing Vertex 7: Step 3



- Now the only vertex v in the bucket of $parent(7) = 6$ is inspected.
- We set $idom(7) = 6$.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\emptyset$	-	1	-
2	1	$\emptyset$	-	2	-
3	2	$\emptyset$	-	3	-
4	3	$\emptyset$	-	4	-
5	4	$\emptyset$	-	5	-
6	2	$\emptyset$	-	6	-
7	6	$\emptyset$	6	6	6

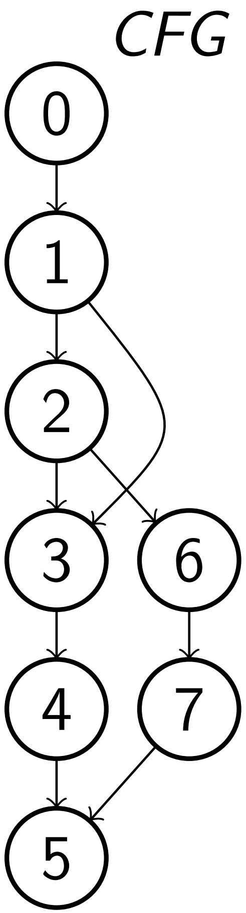
Processing Vertex 6: Step 2



- The only predecessor of $w = 6$ is $v = 2$ which evaluates to $u = 2$.
- $sdom(w = 6)$ becomes 2, and 6 is added to the bucket of $sdom(6) = 2$.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\emptyset$	-	1	-
2	1	$\{6\}$	-	2	-
3	2	$\emptyset$	-	3	-
4	3	$\emptyset$	-	4	-
5	4	$\emptyset$	-	5	-
6	2	$\emptyset$	2	2	-
7	6	$\emptyset$	6	6	6

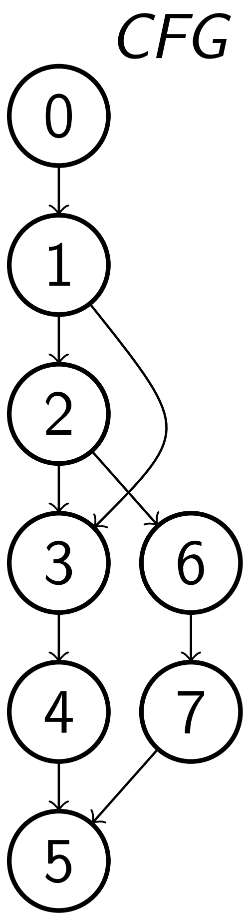
Processing Vertex 6: Step 3



- The bucket of 2 is emptied and *idom*(6) is set to 2.

vertex	parent	bucket	ancestor	sdom	idom
0	-	∅	-	0	-
1	0	∅	-	1	-
2	1	∅	-	2	-
3	2	∅	-	3	-
4	3	∅	-	4	-
5	4	∅	-	5	-
6	2	∅	2	2	2
7	6	∅	6	6	6

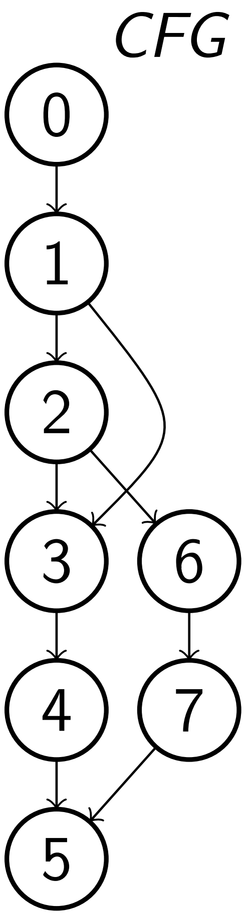
Processing Vertex 5: Step 2



- 5 has two predecessors, 4 and 7.
- After having evaluated 4, $sdom(w = 5)$ tentatively becomes 4.
- Then $eval(7) = 6$ and $sdom(6) = 2$, so the final value of $sdom(w = 5)$ becomes 2, and 5 is added to the bucket of 2.

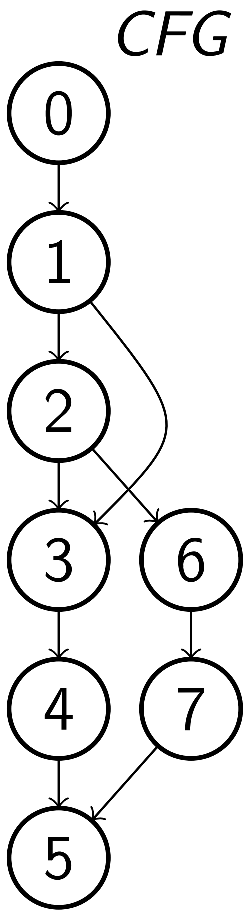
vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\emptyset$	-	1	-
2	1	$\{5\}$	-	2	-
3	2	$\emptyset$	-	3	-
4	3	$\emptyset$	-	4	-
5	4	$\emptyset$	4	2	-
6	2	$\emptyset$	2	2	2
7	6	$\emptyset$	6	6	6

Processing Vertex 4: Step 2



- We find $sdom(4) = 3$, and add 4 to the bucket of 3.

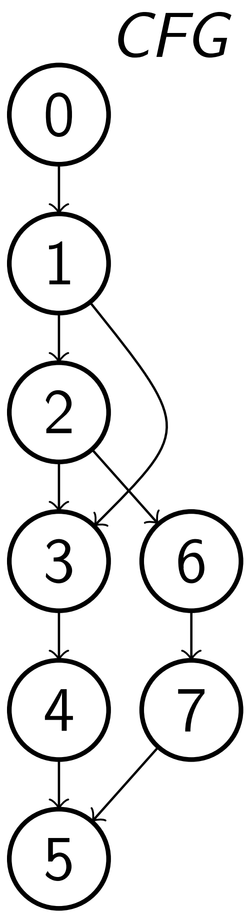
vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\emptyset$	-	1	-
2	1	$\{5\}$	-	2	-
3	2	$\{4\}$	-	3	-
4	3	$\emptyset$	3	3	-
5	4	$\emptyset$	4	2	-
6	2	$\emptyset$	2	2	2
7	6	$\emptyset$	6	6	6



- We set $idom(4) = 3$.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\emptyset$	-	1	-
2	1	$\{5\}$	-	2	-
3	2	$\emptyset$	-	-	-
4	3	$\emptyset$	3	3	3
5	4	$\emptyset$	4	2	-
6	2	$\emptyset$	2	2	2
7	6	$\emptyset$	6	6	6

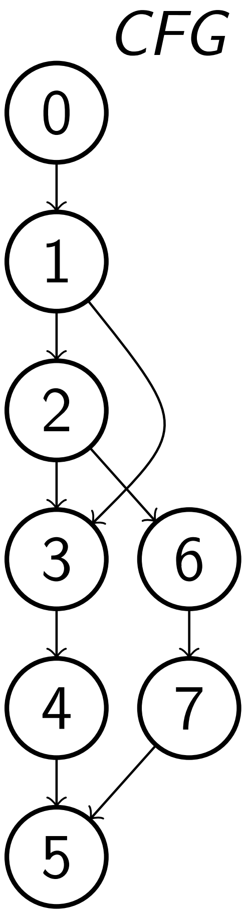
Processing Vertex 3: Step 2



- We find $sdom(3) = 1$, and add 3 to the bucket of 1.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\{3\}$	-	1	-
2	1	$\{5\}$	-	2	-
3	2	$\emptyset$	2	1	-
4	3	$\emptyset$	3	3	3
5	4	$\emptyset$	4	2	-
6	2	$\emptyset$	2	2	2
7	6	$\emptyset$	6	6	6

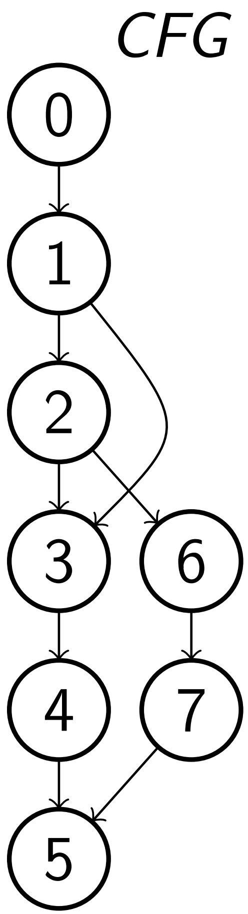
Processing Vertex 3: Step 3



- Now we will empty the bucket of 2 which contains 5.
- *eval*(5) = 3 and *sdom*(3) = 1 < 2, which says there is a path from 0 to 5 which does not include 2. We therefore set *idom*(5) = 3.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\{3\}$	-	1	-
2	1	$\emptyset$	-	2	-
3	2	$\emptyset$	2	1	-
4	3	$\emptyset$	3	3	3
5	4	$\emptyset$	4	2	3
6	2	$\emptyset$	2	2	2
7	6	$\emptyset$	6	6	6

Processing Vertex 2: Step 2

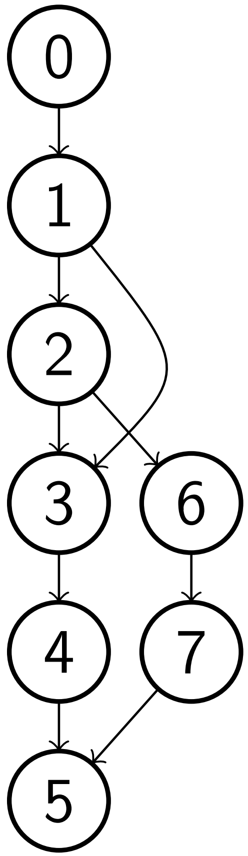


- We find $sdom(2) = 1$, and add 2 to the bucket of 1.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\{2, 3\}$	-	1	-
2	1	$\emptyset$	1	1	-
3	2	$\emptyset$	2	1	-
4	3	$\emptyset$	3	3	3
5	4	$\emptyset$	4	2	3
6	2	$\emptyset$	2	2	2
7	6	$\emptyset$	6	6	6

Processing Vertex 2: Step 3

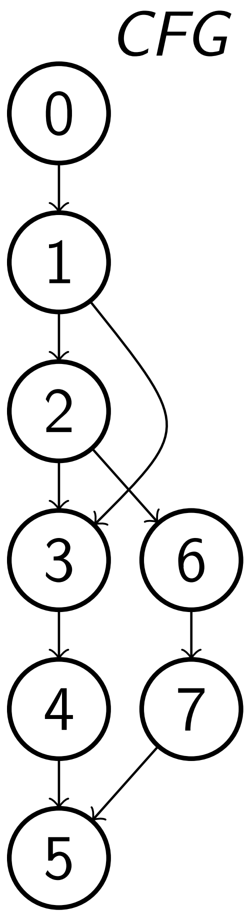
CFG



- Now we will empty the bucket of 1 which contains 2 and 3, both of which find 1 to be their immediate dominator.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\emptyset$	-	1	-
2	1	$\emptyset$	1	1	1
3	2	$\emptyset$	2	1	1
4	3	$\emptyset$	3	3	3
5	4	$\emptyset$	4	2	3
6	2	$\emptyset$	2	2	2
7	6	$\emptyset$	6	6	6

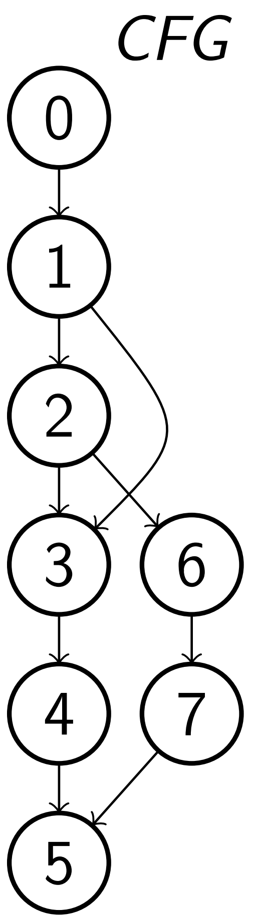
Processing Vertex 1: Step 2



- Finally, we find $sdom(1) = 0$.

vertex	parent	bucket	ancestor	sdom	idom
0	-	$\emptyset$	-	0	-
1	0	$\emptyset$	0	0	0
2	1	$\emptyset$	1	1	1
3	2	$\emptyset$	2	1	1
4	3	$\emptyset$	3	3	3
5	4	$\emptyset$	4	2	3
6	2	$\emptyset$	2	2	2
7	6	$\emptyset$	6	6	6

After Step 4



vertex	parent	bucket	ancestor	sdom	idom
0	-	∅	-	0	-
1	0	∅	0	0	0
2	1	∅	1	1	1
3	2	∅	2	1	1
4	3	∅	3	3	3
5	4	∅	4	2	1
6	2	∅	2	2	2
7	6	∅	6	6	6