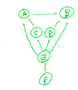
CSE 421
Introduction to Algorithms

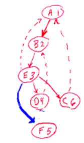
Depth First Search and
Strongly Connected Components

1

Undirected
Depth-First Search



- It's not just for trees

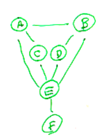


```
DFS(v)
back edge {if v marked then return;
tree edge {mark v; #v := ++count;
          {for all edges (v,w) do DFS(w);
Main()
    count := 0;
    for all unmarked v do DFS(v);
```

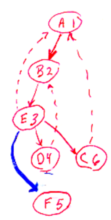
NB: book uses decreasing

2

Undirected
Depth-First Search



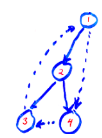
- Key Properties:
  1. No "cross-edges";
     only tree- or back-edges
  2. Before returning, DFS(v) visits all vertices reachable from v via paths through previously unvisited vertices



3

Directed Depth First Search

- Algorithm: Unchanged
- Key Properties:
  2. Unchanged
  1'. Edge (v,w) is:



As before	Tree-edge	if w unvisited
	Back-edge	if w visited, #w<#v, on stack
New	Cross-edge	if w visited, #w<#v, not on stack
	Forward-edge	if w visited, #w>#v

Note: Cross edges *only* go "Right" to "Left"

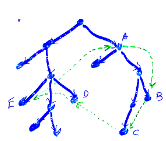
4

An Application:

G has a cycle ⇔ DFS finds a back edge

⇐ Clear.

⇒ Why can't we have something like this?:



5

Lemma 1

Before returning, dfs(v) visits
- all unvisited vertices reachable from v
- only unvisited vertices reachable from v

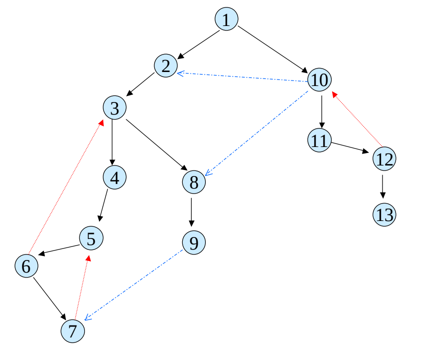
All become descendants of v in the tree.

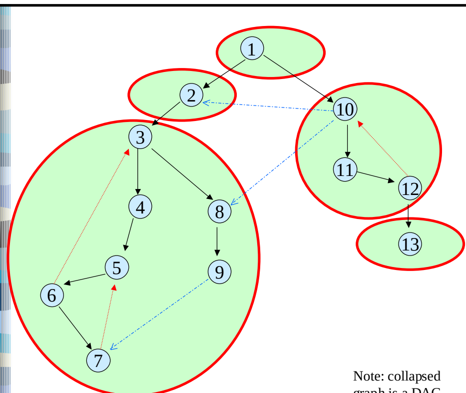
Proof:
- dfs follows all direct out-edges
- call dfs recursively at each
- by induction on path length, visits all

6

Strongly Connected Components

- Defn: G is *strongly connected* if for all u,v there is a (directed) path from u to v and from v to u.
  [Equivalently:
  there is a cycle through u and v.]
- Defn: a *strongly connected component* of G is a maximal strongly connected subgraph.





Note: collapsed graph is a DAG

Uses for SCC's

- Optimizing compilers need to find loops, which are SCC's in the program flow graph.
- Nontrivial SCC's in call-graph are sets of mutually recursive procedures
- If (u,v) means process u is waiting for process v, SCC's show deadlocks.

Two Simple SCC Algorithms

- u,v in same SCC iff there are paths $u \to v$ & $v \to u$
- Transitive closure: $O(n^3)$
- DFS from every u, v: $O(ne) = O(n^3)$

Goal:

- Find all Strongly Connected Components in linear time, i.e., time $O(n+e)$

  (Tarjan, 1972)

Definition

The *root* of an SCC is the first vertex in it visited by DFS.

Equivalently, the root is the vertex in the SCC with the smallest number.

13

Lemma 2

All members of an SCC are descendants of its root.

Proof:

- all members are reachable from all others
- so, all are reachable from its root
- all are unvisited when root is visited
- so, all are descendants of its root (Lemma 1)

14

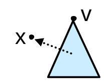
Subgoal

- Can we identify some root?
- How about the root of the first SCC completely explored by DFS?

- Key idea: no exit from first SCC
(first SCC is leftmost “leaf” in collapsed DAG)

15

Definition

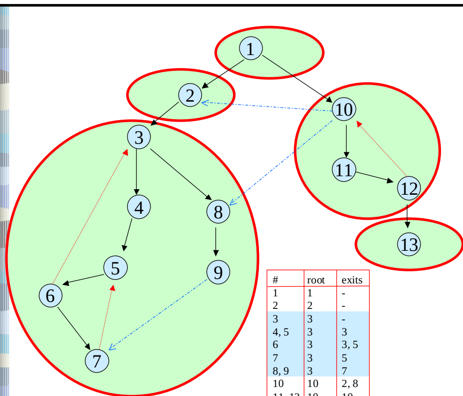


x is an *exit* from v (from v’s subtree) if

- x is not a descendant of v, but
- x is the head of a (cross- or back-) edge from a descendant of v (including v itself)

NOTE: #x < #v

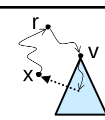
16



#	root	exits
1	1	-
2	2	-
3	3	-
4, 5	3	3
6	3	3, 5
7	3	5
8, 9	3	7
10	10	2, 8
11, 12	10	10
13	13	-

17

Lemma 3



If v is not a root, then v has an exit.

Proof:

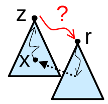
- let r be root of v’s SCC
- r is a proper ancestor of v (Lemma 2)
- let x be the first vertex that is not a descendant of v on a path v → r .
- x is an exit

Cor: If v has no exit, then v is a root.

NB: converse not true; some roots do have exits

18

Lemma 4



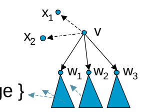
If r is the first root from which dfs returns, then r has no exit

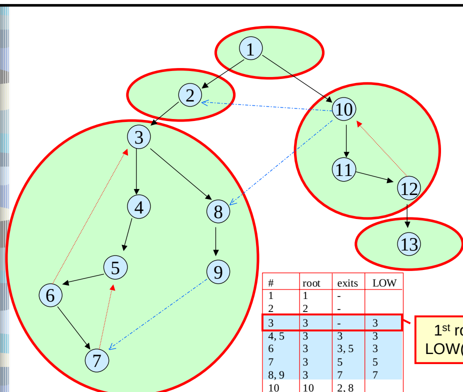
Proof:

- Suppose x is an exit
- let z be root of x's SCC
- r not reachable from z, else in same SCC
- #z ≤ #x (z ancestor of x; Lemma 2)
- #x < #r (x is an exit from r)
- #z < #r, no z → r path, so return from z first
- Contradiction

How to Find Exits (in 1st component)

- All exits x from v have #x < #v
- Suffices to find any of them, e.g. min #
- Defn:
  LOW(v) = min({ #x | x an exit from v} ∪ {#v})
- Calculate inductively:
  LOW(v) = min of:
  - #v
  - { LOW(w) | w a child of v }
  - { #x | (v,x) is a back- or cross-edge }
- 1st root : LOW(v)=v





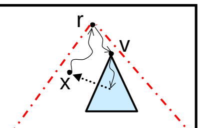
#	root	exits	LOW
1	1	-	
2	2	-	
3	3	-	3
4, 5	3	3	3
6	3	3, 5	3
7	3	5	5
8, 9	3	7	7
10	10	2, 8	
11, 12	10	10	
13	13	-	

1st root: LOW(v)=v

Finding Other Components

- Key idea: No exit from
  - 1st SCC
  - 2nd SCC, except maybe to 1st
  - 3rd SCC, except maybe to 1st and/or 2nd
  - ...

Lemma 3'



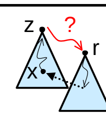
If v is not a root, then v has an exit in v's SCC.

Proof:

- let r be root of v's SCC
- r is a proper ancestor of v (Lemma 2)
- let x be the first vertex that is not a descendant of v on a path v → r .
- x is an exit in v's SCC

Cor: If v has no exit in v's SCC, then v is a root.

Lemma 4'



If r is the ~~first~~ kth root from which dfs returns, then r has no exit except possibly to the first (k-1) components

Proof:

- Suppose x is an exit
- let z be root of x's SCC
- r not reachable from z, else in same SCC
- #z ≤ #x (z ancestor of x; Lemma 2)
- #x < #r (x is an exit from r)
- #z < #r, no z → r path, so return from z first
- ~~Contradiction~~ i.e., x in first (k-1)

How to Find Exits (in ~~1st~~ k^{th} component)

- All exits x from v have #x < #v
- Suffices to find any of them, e.g. min #
- Defn:
  LOW(v) = min({ #x | x an exit from v } ∪ {#v})
  and x not in first (k-1) components
- Calculate inductively:
  LOW(v) = min of:
  - #v
  - { LOW(w) | w a child of v }
  - { #x | (v,x) is a back- or cross-edge }

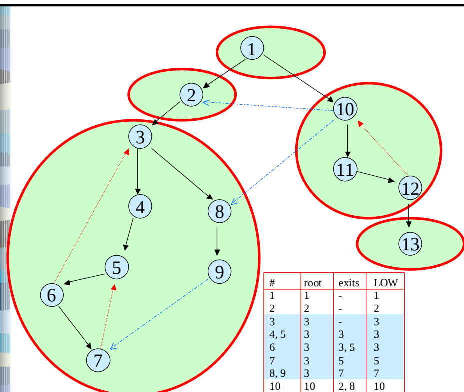
25

SCC Algorithm

#v = DFS number
v.low = LOW(v)
v.scc = component #

```
SCC(v)
  #v = vertex_number++; v.low = #v; push(v)
  for all edges (v,w)
    if #w == 0 then
      SCC(w); v.low = min(v.low, w.low) // tree edge
    else if #w < #v && w.scc == 0 then
      v.low = min(v.low, #w)     // cross- or back-edge
  if #v == v.low then            // v is root of new scc
    scc#++;
    repeat
      w = pop(); w.scc = scc#;   // mark SCC members
    until w==v
```

26



#	root	exits	LOW
1	1	-	1
2	2	-	2
3	3	-	3
4, 5	3	3	3
6	3	3, 5	3
7	3	5	5
8, 9	3	7	7
10	10	2, 8	10
11, 12	10	10	10
13	13	-	13

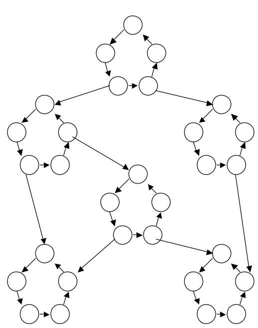
27

Complexity

- Look at every edge once
- Look at every vertex (except via in-edge) at most once
- Time = O(n+e)

28

Example



29