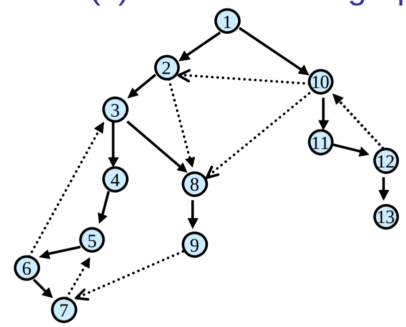
CSE 417: Algorithms and Computational Complexity

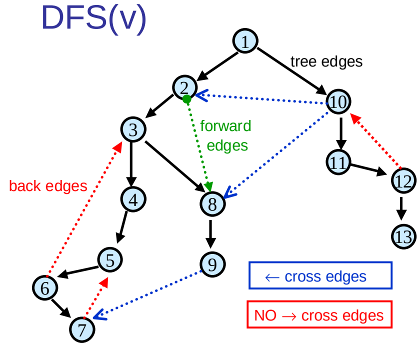
Winter 2001

DFS and Strongly Connected Components

DFS(v) for a directed graph



DFS(v)



Properties of Directed DFS

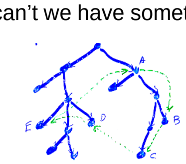
- Before DFS(v) returns, it visits all previously unvisited vertices reachable via directed paths from v

An Application:

G has a cycle ⇔ DFS finds a back edge

⇐ Clear.

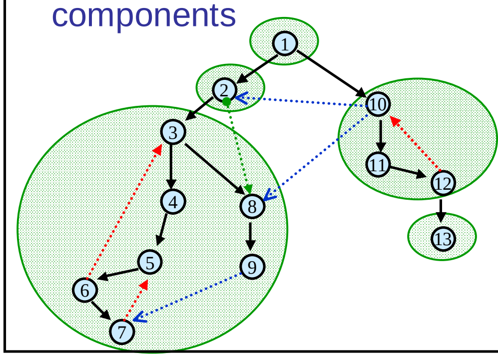
⇒ Why can't we have something like this?:



Strongly-connected components

- In directed graph if there is a path from a to b there might not be one from b to a
- a and b are **strongly connected** iff there is a path in both directions (i.e. a directed cycle containing both a and b
- Breaks graph into components

Strongly-connected components



7

Uses for SCC's

- Optimizing compilers:
  - SCC's in the program flow graph = "loops"
  - SCC's in call-graph = mutually recursive procedures
- Operating systems: If (u,v) means process u is waiting for process v, SCC's show deadlocks.
- Econometrics: SCC's might show highly interdependent sectors of the economy

8

Directed Acyclic Graphs

- If we collapse each SCC to a single vertex we get a directed graph with no cycles
  - a **directed acyclic graph** or **DAG**
- Many problems on directed graphs can be solved as follows:
  - Compute SCC's and resulting DAG
  - Do one computation on each SCC
  - Do another computation on the overall DAG

9

Simple SCC Algorithm

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

10

Better method

- Can compute all the SCC's while doing a single DFS! $O(n+m)$ time
- We won't do the full algorithm but will give some ideas

11

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 in DFS ordering.

Fact: All members of an SCC are descendants (via tree edges) of its root.

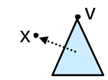
Exercise: show that each SCC is a *contiguous* subtree.

12

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)

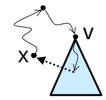
Definition



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)

- Any non-root vertex v has an exit

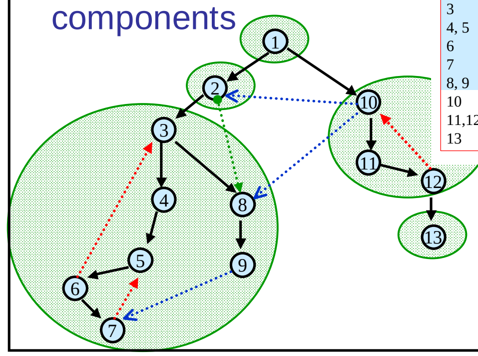


Finding SCC's

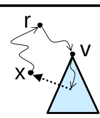
- Root nodes v sometimes have exits
  - *But* only via a cross-edge to a node x that is not in a component with a root above v, e.g. vertex 10 in the example.

Strongly-connected components

#	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	-



Non-Roots Have Exits
(Idea: on cycle back to root)



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

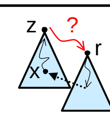
Proof:

- let r be root of v's SCC
- r is a proper ancestor of v (Fact about roots)
- let x be the first vertex that is not a descendant of v on a path v $\rightarrow$ 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

First Root: Exit-less
(Idea: exit ➔ bigger cycle)



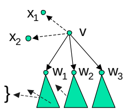
If r is the first root from which dfs returns, then r has no exit

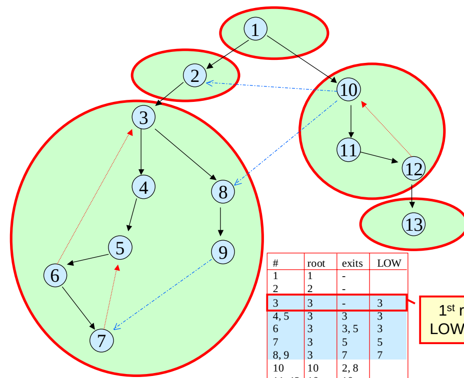
Proof (by contradiction):

- 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; Fact about roots)
- #x < #r (x is an exit from r)
- #z < #r, no z $\rightarrow$ r path, so return from z first
- Contradiction

How to Find Exits
(from 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





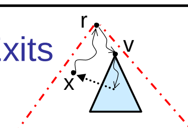
#	root	exits	LOW
1	1	-	
2	2	-	
3	3	-	3
4, 5	3	3	3
6	3	3, 5	3
7	3	5	5
8, 9	3	7	7
10	10	2, 8	
11, 12	10	10	
13	13	-	

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
  - ...

Non-Roots Have Exits (Revisited)



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

in v's SCC

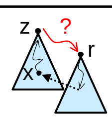
Proof:

- let r be root of v's SCC
- r is a proper ancestor of v (Fact about roots)
- 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 , then v is a root.

in v's SCC

First Root: Exit-less (Revisited) kth



If r is the ~~first~~ 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; Fact about roots)
- #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~~ kth component)

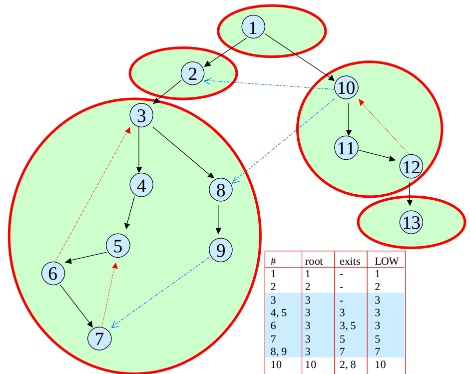
- All exits x from v have #x < #v
- Suffices to find any of them, e.g. min #
- Defn:
  LOW(v) = min({ #x | x an exit from v} ∪ {#v})
- Calculate inductively:
  LOW(v) = min of:
  - #v
  - { LOW(w) | w a child of v }
  - { #x | (v,x) is a back- or cross-edge }

and x not in first (k-1) components

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
```

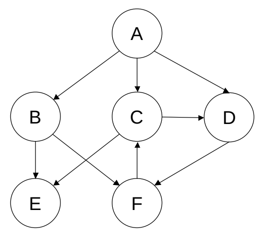



#	root	exits	LOW
1	1	-	1
2	2	-	2
3	3	-	3
4, 5	3	3	3
6	3	3, 5	3
7	3	5	5
8, 9	3	7	7
10	10	2, 8	10
11, 12	10	10	10
13	13	-	13

Complexity

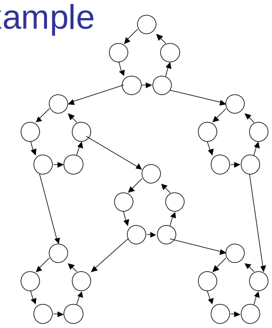
- Look at every edge once
- Look at every vertex (except via in-edge) at most once
- Time = O(n+e)

Example



dfs#	v	root	exits	low(v)
1				
2				
3				
4				
5				
6				

Example



v	Low(v)	v	Low(v)