CSE 417: Algorithms and Computational Complexity

Graphs & Graph Algorithms II

Autumn 2002
Paul Beame

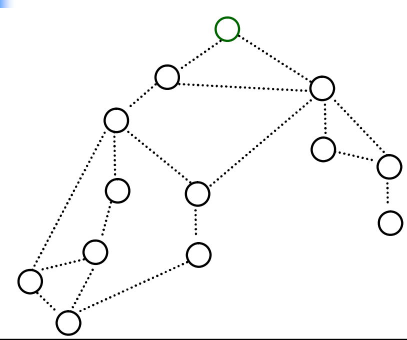
Depth-First Search

- Follow the first path you find as far as you can go
- Back up to last unexplored edge when you reach a dead end, then go as far you can
- Naturally implemented using recursive calls or a stack

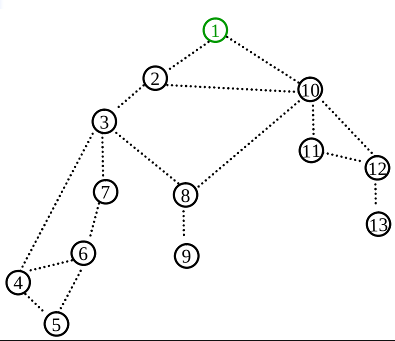
DFS(v) – Recursive version

Global Initialization: mark all vertices "undiscovered"
DFS(**v**)
  mark **v** "discovered"
  for each edge {**v**,**x**}
    if (x is "undiscovered")
      DFS(**x**)
  end for
  mark **v** "fully-explored"

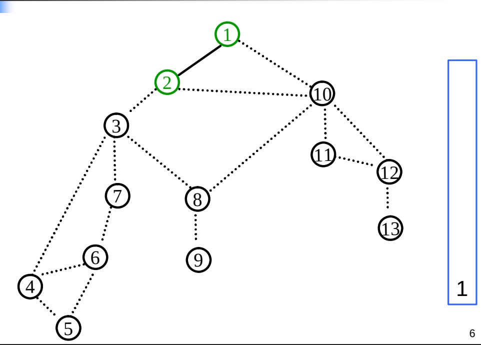
DFS(v)



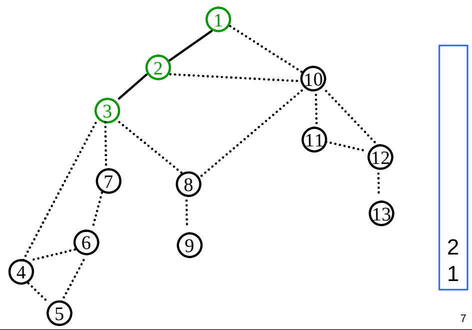
DFS(v)



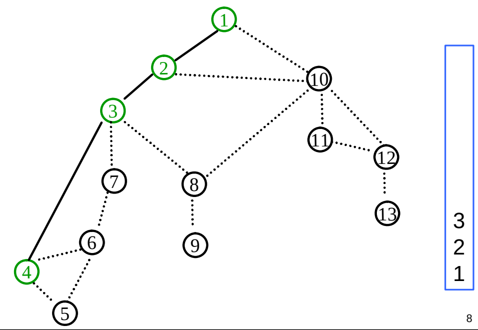
DFS(v)



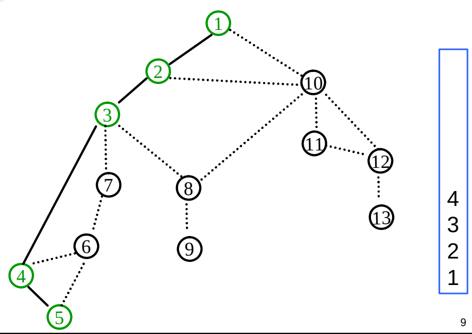
DFS(v)



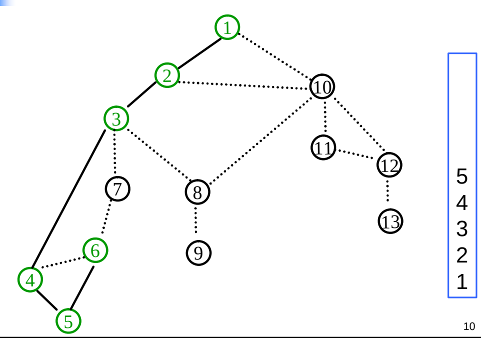
DFS(v)



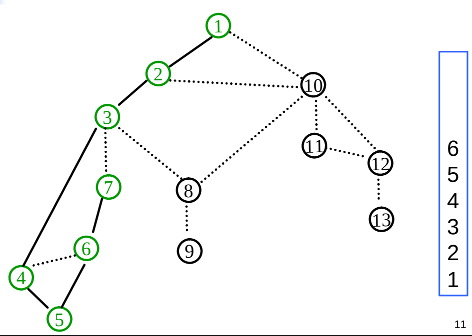
DFS(v)



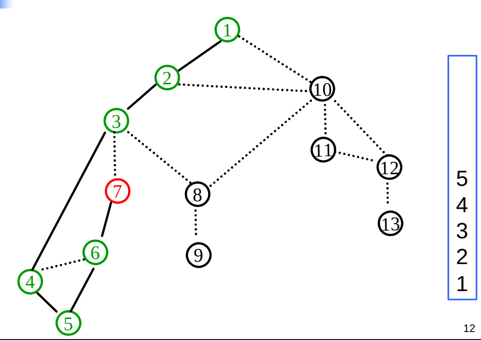
DFS(v)



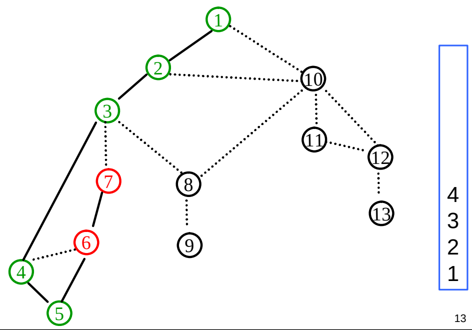
DFS(v)



DFS(v)

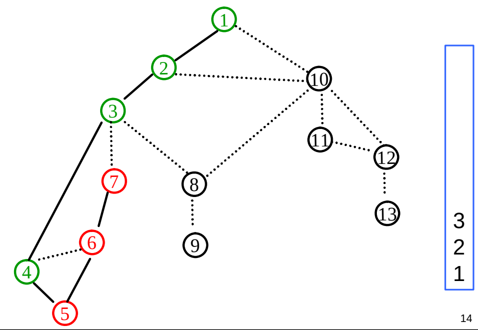


DFS(v)



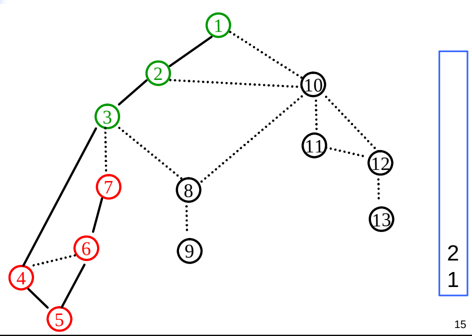
13

DFS(v)



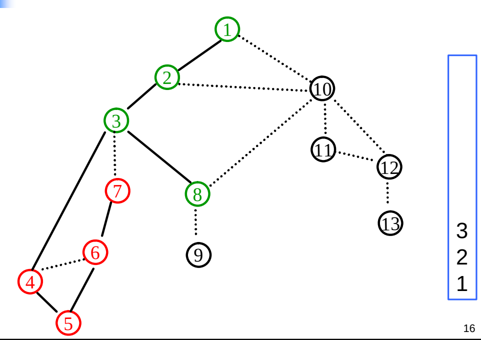
14

DFS(v)



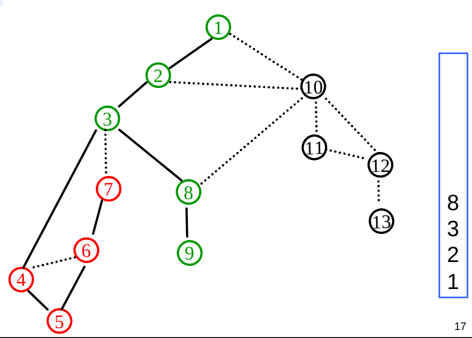
15

DFS(v)



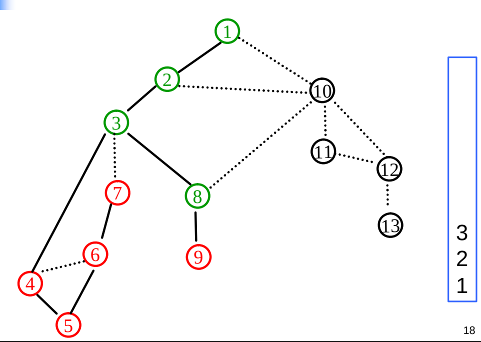
16

DFS(v)



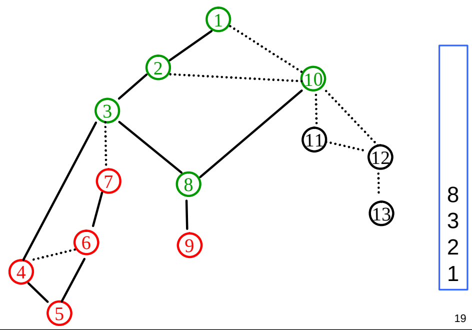
17

DFS(v)

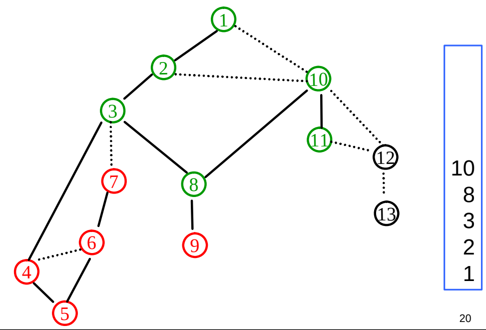


18

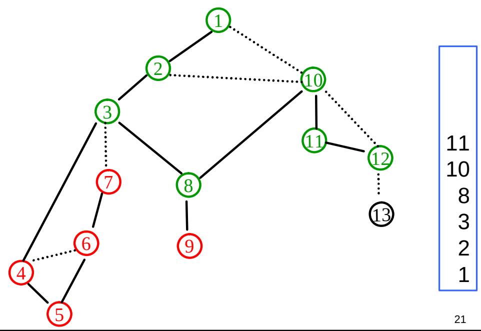
DFS(v)



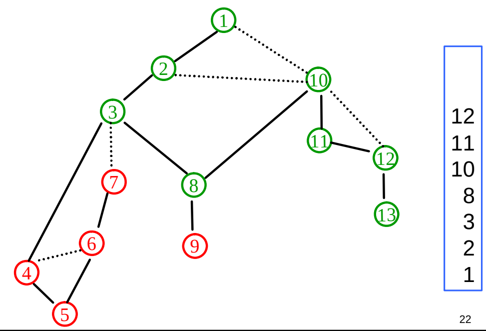
DFS(v)



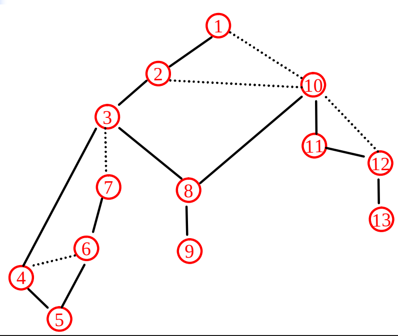
DFS(v)



DFS(v)



DFS(v)

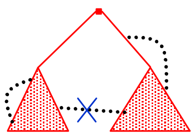


Properties of DFS(v)

- Like BFS(**v**):
  - DFS(**v**) visits **x** if and only if there is a path in **G** from **v** to **x**
  - Edges into undiscovered vertices define a tree
    - "depth first spanning tree" of **G**
- Unlike the BFS tree:
  - the DFS spanning tree isn't minimum depth
  - its levels don't reflect min distance from the root
  - non-tree edges never join vertices on the same or adjacent levels
- BUT…

Non-tree edges

- All non-tree edges join a vertex and one of its descendents/ancestors in the DFS tree
- No cross edges!



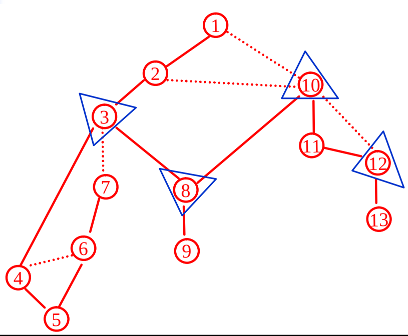
25

Application: Articulation Points

- A node in an undirected graph is an **articulation point** iff removing it disconnects the graph
- articulation points represent vulnerabilities in a network – single points whose failure would split the network into 2 or more disconnected components

26

Articulation Points



27

Articulation Points from DFS

- Non-tree edges eliminate articulation points
- Root node is articulation point ⇔ it has more than one child
- Leaf nodes are never articulation points
- Other nodes **u** are articulation points ⇔
  - no non-tree edges going from some child of **u** to above **u** in the tree

28

Articulation Points from DFS

- For each vertex **v** compute
  - small(**v**)
    - the smallest number of a node pointed at by any descendant of **v** in the DFS tree (including **v** itself)
  - Can compute small(v) for every **v** during DFS at minimal extra cost
- Non-leaf, non-root node **u** is an articulation point ⇔ for some child **v** of **u**
  - small(**v**) = DFSnumber(**u**)
  - Easy to compute and check during DFS

29

DFS(v) – Recursive version

Global Initialization:
  mark all vertices **u** "undiscovered" via **dfsnum**[**u**] ← **-1**
  dfscounter ← **0**

DFS(**v**)
  dfscounter ← **dfscounter+1**
  dfsnum[**v**] ← **dfscounter**   // mark **v** "discovered"
  for each edge (**v**,**x**)
    if (**dfsnum**[**x**] = **-1**)   // **x** previously undiscovered
      add edge (**v**,**x**) to DFStree
      DFS(**x**)
  // mark **v** "fully-explored"

30

DFS(v) for Finding Articulation Points

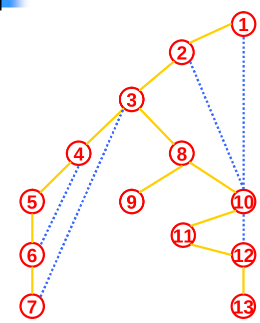
```
Global initialization: dfsnum[u] ← -1 for all u; dfscounter ←0
DFS(v)
 dfscounter ← dfscounter+1
 dfsnum[v] ← dfscounter
 small[v] ← dfsnum[v]          // initialization
 for each edge {v,x}
      if (dfsnum[x] = -1)      // x is undiscovered
          DFS(x)
          if (small[x] >= dfsnum[v])
              print "v is an articulation point, separating x"
          small[v] ← min(small[v], small[x])
      else if (x is not v's parent)
          small[v] ← min(small[v], dfsnum[x])
```

Check that {v,x} is a back edge (not a tree edge)

Note: need a separate check for the root

31

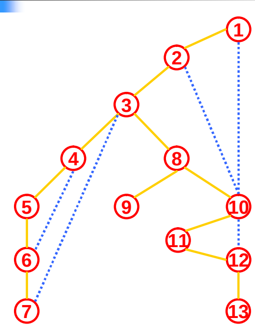
Articulation Points



DFS #	Small
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	

32

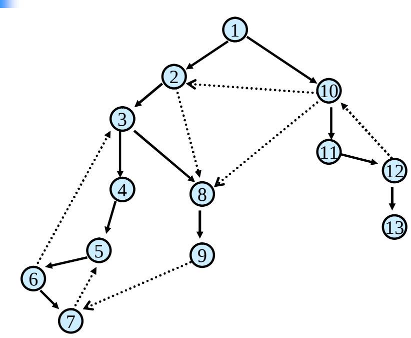
Articulation Points



DFS #	Small	Art
1	1	
2	1	
3	1	Y
4	3	
5	3	
6	3	
7	3	
8	1	Y
9	9	
10	1	Y
11	10	
12	10	Y
13	13	

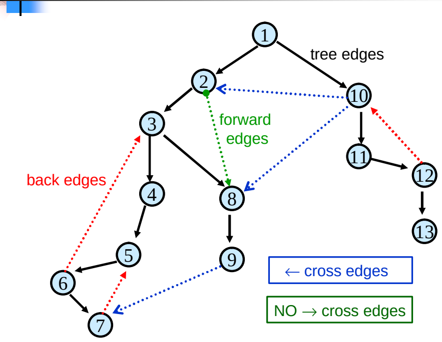
33

DFS(v) for a directed graph



34

DFS(v)



35

Properties of Directed DFS

- Before DFS(v) returns, it visits all previously unvisited vertices reachable via directed paths from v
- Every cycle contains a back edge in the DFS tree

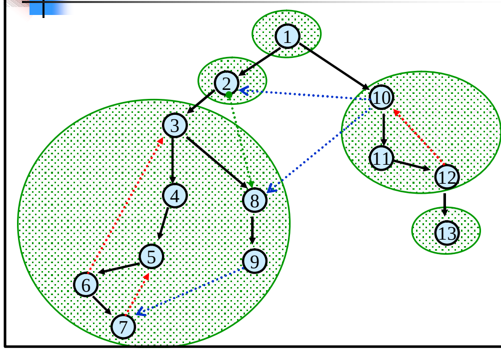
36

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

37

Strongly-connected components



38

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

39

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

40

Simple SCC Algorithm

- **u**,**v** in same SCC iff there are paths **u** $\rightarrow$ **v** & **v** $\rightarrow$ **u**
- DFS/BFS from every **u**, **v**:
  - Time **O**(**nm**) = **O**(**n**^{3})

41

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

42

Definition

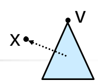
The **root** of an SCC is the first vertex in it visited by DFS.

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

Subgoal

- All members of an SCC are descendants of its root.
- 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

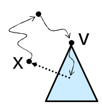
Definition



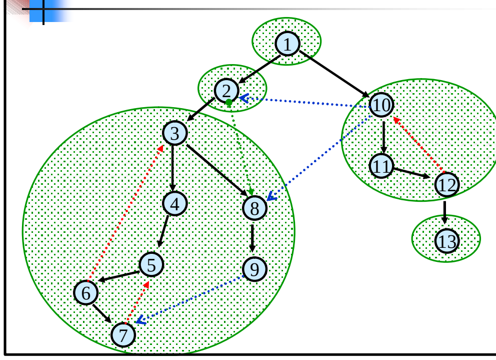
x is an *exit* from v (from v's subtree) if

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

- Any non-root vertex v has an exit



Strongly-connected components



Finding Other Components

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

SCC Algorithm

scc[v] = component #

```
SCC(v)
   dfsnum[v] ← dfscounter++;
   small[v] ← dfsnum[v]
   push(v)
   for all edges (v,w)
      if dfsnum[w] = -1 then
         SCC(w)
         small[v] ← min(small[v], small[w]) // tree edge
      else if dfsnum[w] < dfsnum[v] and scc[w] = 0 then
         small[v] ← min(small[v], dfsnum[w])      // cross- or back-edge
   if dfsnum[v ] = small[v] then                  // v is root of new scc
      sccnum←sccnum+1;
      repeat
         w = pop(); scc[w] = sccnum;       // mark SCC members
      until w=v
```