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”
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 descendent/ancestors in the DFS tree
- No cross edges!

**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

**Articulation Points**
- For each vertex \( v \) compute:
  - \( \text{small}(v) \)
    - the smallest number of a node pointed at by any descendant of \( v \) in the DFS tree (including \( v \) itself)
    - Can compute \( \text{small}(v) \) for every \( v \) during DFS at minimal extra cost
  - Non-leaf, non-root node \( u \) is an articulation point \( \iff \)
    - \( \text{small}(v) = \text{DFSNumber}(u) \)
    - Easy to compute and check during DFS

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

**Articulation Points from DFS**

**DFS(\( v \)) – Recursive version**

Global Initialization:
- mark all vertices \( u \) “undiscovered” via \( \text{dfsnum}(u) \leftarrow -1 \)
- \( \text{dfscounter} \leftarrow 0 \)

\( \text{DFS}(v) \)
- \( \text{dfscounter} \leftarrow \text{dfscounter} + 1 \)
- \( \text{dfsnum}(v) \leftarrow \text{dfscounter} \) // mark \( v \) “discovered”
- for each edge \( (v,x) \)
  - if \( \text{dfsnum}(x) = -1 \) // \( x \) previously undiscovered
    - add edge \( (v,x) \) to DFS tree
    - \( \text{DFS}(x) \) // mark \( v \) “fully-explored”
**DFS(v) for Finding Articulation Points**

Global initialization: \( \text{dfsnum}(u) \leftarrow -1 \) for all \( u \); \( \text{dfscounter} \leftarrow 0 \)

- \( \text{dfscounter} \leftarrow \text{dfscounter} + 1 \)
- \( \text{dfsnum}(v) \leftarrow \text{dfscounter} \) // initialization

for each edge \((v, x)\)

- if \( \text{dfsnum}(x) = -1 \) // \( x \) is undiscovered
  - DFS(x)
  - if \( \text{small}(x) \geq \text{dfsnum}(v) \)
    - print "\( v \) is an articulation point, separating \( x \)"
  - else if \( x \) is not \( v \)'s parent
    - \( \text{small}(v) \leftarrow \min(\text{small}(v), \text{dfsnum}(x)) \)

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

Note: need a separate check for the root

---

**Articulation Points**

<table>
<thead>
<tr>
<th>DFS #</th>
<th>Small</th>
<th>Art</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

---

**DFS(v) for a directed graph**

**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
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

Uses for SCC’s

- Optimizing compilers:
  - SCC’s in 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

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

Simple SCC Algorithm

- u, v in same SCC iff there are paths u → v & v → u
- DFS/BFS from every u, v:
  - Time $O(nm) = O(n^2)$

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
**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**

- **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
  - 1st SCC
  - 2nd SCC, except maybe to 1st
  - 3rd SCC, except maybe to 1st and/or 2nd
  - ...

**SCC Algorithm**

```
scc[v] = component #

SCC(v)
  dfsnum[v] = dfnnum[v] = -1
  small[v] = dfsnum[v] = -1
  for all edges (v, w)
    if dfsnum[w] = -1
      SCC(w)
      small[v] = min(small[v], small[w]) // tree edge
    else if dfsnum[w] < dfsnum[v] and scc[w] = 0
      small[v] = dfsnum[w] // cross- or back-edge
      if dfsnum[v] = small[v] then
        sccnum--
        sccnum = sccnum + 1
      end if
      scc[w] = sccnum
    end if
  end for
  if dfsnum[v] = small[v] then
    sccnum--
    sccnum = sccnum + 1
    repeat
    w = pop()
    scc[w] = sccnum
  end if
```