DFS and Strongly Connected Components

**Properties of Directed DFS**
- Before DFS(v) returns, it visits all previously unvisited vertices reachable via directed paths from v.

**Strongly-connected components**
- In a 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.

**An Application:**
G has a cycle ⇔ DFS finds a back edge
⇔ Clear.
⇒ Why can't we have something like this?:

G has a cycle ⇔ DFS finds a back edge
⇔ Clear.
⇒ Why can't we have something like this?:
Strongly-connected components

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

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 \(u \rightarrow v \& v \rightarrow u\)
- DFS from every \(u, v\): \(O(nm) = O(n^3)\)

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.

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

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

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** → **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 **r**’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
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:
  \[ \text{LOW}(v) = \min(\#x | x \text{ an exit from } v) \cup \{#v\} \]
- Calculate inductively:
  \[ \text{LOW}(v) = \min \text{ of:} \]
  - #v
  - \{ \text{LOW}(w) | w \text{ a child of } v \}
  - \{ #x | (v,x) \text{ is a back- or cross-edge} \}
- 1st root : \text{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 .

Proof:
- let r be root of v’s SCC
- r is a proper ancestor of v
- 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.

First Root: Exit-less (Revisited)

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 v’s SCC
- r not reachable from z, else in same SCC
- #z \leq #x (z ancestor of x; Fact about roots)
- #z < #r (x is an exit from r)
- #z < #r, no z \rightarrow r path, so return from z first
- Contradiction
- i.e., x in first (k-1) components

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:
  \[ \text{LOW}(v) = \min(\#x | x \text{ an exit from } v) \cup \{#v\} \]
- Calculate inductively:
  \[ \text{LOW}(v) = \min \text{ of:} \]
  - #v
  - \{ \text{LOW}(w) | w \text{ a child of } v \}
  - \{ #x | (v,x) \text{ is a back- or cross-edge} \}

i.e., x in first (k-1) components

and x not in first (k-1) components
**SCC Algorithm**

\[ \text{SCC}(v) \]

\[ \text{v} = \text{vertex\_number}++; \quad \text{v}.\text{low} = \text{#v}; \quad \text{push}(v) \]

for all edges \((v, w)\)

- if \(#w == 0\) then
  \[ \text{SCC}(w); \quad \text{v}.\text{low} = \min(\text{v}.\text{low}, \text{w}.\text{low}) \]  // tree edge

- else if \(#v < \#w \&\& w.\text{scc} == 0\) then
  \[ \text{v}.\text{low} = \min(\text{v}.\text{low}, \#w) \]  // cross- or back-edge

- if \(#v == \text{v}.\text{low}\) then  // \(v\) is root of new SCC
  \[ \text{scc}\#++; \]

repeat

\[ w = \text{pop}(); \quad w.\text{scc} = \text{scc}\#; \]  // mark SCC members

until \(w == v\)

\[ \text{#v} = \text{DFS number} \]

\[ \text{v}.\text{low} = \text{LOW}(v) \]

\[ \text{v}.\text{scc} = \text{component \#} \]

**Complexity**

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

**Example**

<table>
<thead>
<tr>
<th>dfs#</th>
<th>v</th>
<th>root</th>
<th>exits</th>
<th>low(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**Example**