CSE 521
Algorithms

Depth First Search and
Strongly Connected Components

W.L. Ruzzo, Winter 2013
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
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
  - 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”
An Application:

\[ G \text{ has a cycle} \iff \text{DFS finds a back edge} \]
\[ \iff \text{Easy - back edge } (x, y) \text{ plus tree edges } y, \ldots, x \text{ form a cycle.} \]
\[ \Rightarrow \text{Why can’t we have something like this?:} \]
Lemma 1

Before returning, \( \text{dfs}(v) \) visits \( w \) iff
- \( w \) is unvisited
- \( w \) is reachable from \( v \) via a path through unvisited vertices

Proof sketch:
- \( \text{dfs} \) follows all direct out-edges
- call \( \text{dfs} \) recursively at each unvisited one
- use induction on \# of such \( w \)
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 circuit through \( u \) and \( v \).]

- **Defn:** a *strongly connected component* of G is a maximal strongly connected (vertex-induced) subgraph.
Note: collapsed graph is a DAG
Uses for SCC’s

- Optimizing compilers:
  - SCC’s in program flow graph = loops
  - SCC’s in call graph = mutual recursion

- Operating Systems: If (u,v) means process u is waiting for process v, SCC’s show deadlocks.

- Spreadsheet eval: circular dependencies

- Econometrics: SCC’s might show highly interdependent sectors of the economy.

- Etc.
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 on the overall DAG
  - Example: Spreadsheet evaluation
Two Simple SCC Algorithms

- u,v in same SCC iff there are paths u → v & v → 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 DFS number.
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)

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 (returned from) 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)

**NOTE:** \( #x < #v \)

**Ex:** node #1 cannot have an exit.
Lemma 3: Nonroots have exits

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 $v \rightarrow r$.
- $x$ is an exit

Cor (contrapositive): If $v$ has no exit, then $v$ is a root.

NB: converse not true; some roots do have exits
Lemma 4: No Escaping 1st Root

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; Lemma 2)
- \#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 (in 1\textsuperscript{st} 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:
  \(\text{LOW}(v) = \min \text{ of:}\)
  - \(\#v\)
  - \(\{ \text{LOW}(w) | w \text{ a child of } v \}\)
  - \(\{ \#x | (v,x) \text{ is a back- or cross-edge} \}\)
- 1\textsuperscript{st} root: \(\text{LOW}(v)=v\)
1st root: LOW(v) = v
Finding Other Components

- **Key idea**: No exit from
  - 1\textsuperscript{st} SCC
  - 2\textsuperscript{nd} SCC, except maybe to 1\textsuperscript{st}
  - 3\textsuperscript{rd} SCC, except maybe to 1\textsuperscript{st} and/or 2\textsuperscript{nd}
  - ...
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 $v \rightarrow r$
– $x$ is an exit

Cor: If $v$ has no exit, then $v$ is a root.
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
- \( \#z \leq \#x \) (\( z \) ancestor of \( x \); Lemma 2)
- \( \#x < \#r \) (\( x \) 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

except possibly to the 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:
  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 }

NB: defn of "exit" has not changed, but we’re not interested in exits into previous SCCs
SCC Algorithm

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

#v = DFS number
v.low = LOW(v)
v.scc = component #
Has exits, but is a root

<table>
<thead>
<tr>
<th>#</th>
<th>root</th>
<th>exits</th>
<th>LOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>4, 5</td>
<td>3</td>
<td>3, 5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8, 9</td>
<td>3</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2, 8</td>
<td>10</td>
</tr>
<tr>
<td>11, 12</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>-</td>
<td>13</td>
</tr>
</tbody>
</table>
Complexity

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

- Unlike undirected DFS, start vertex matters
- Add “outer loop”:

  mark all vertices unvisited
  while there is unvisited vertex v do
    scc(v)

- Exercise: redo example starting from another vertex, e.g. #11 or #13 (which become #1)
Example

\[
\begin{array}{cccc}
\text{dfs#} & v & \text{root} & \text{exits} & \text{low}(v) \\
1 & & & & \\
2 & & & & \\
3 & & & & \\
4 & & & & \\
5 & & & & \\
6 & & & & \\
\end{array}
\]

A \rightarrow B \rightarrow E
A \rightarrow C \rightarrow F
A \rightarrow D
B \rightarrow C
C \rightarrow D
E \rightarrow F
Example