CSE 421
Introduction to Algorithms

Depth First Search and
Strongly Connected Components

W.L. Ruzzo, Winter 2004

Undirected Depth-First Search

- It’s not just for trees

DFS(v)
  if v marked then return;
  mark v; #v := ++count;
  for all edges (v,w) do DFS(w);

Main()
  count := 0;
  for all unmarked v do DFS(v);

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:
   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

New

Note: Cross edges only go “Right” to “Left”

An Application:

G has a cycle ↔ DFS finds a back edge
  Clear.
  ⇒ Why can’t we have something like this?:

Lemma 1

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

Proof:
- dfs follows all direct out-edges
- call dfs recursively at each unvisited one
- by induction on path length, visits all
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 cycle through u and v.]
- **Defn:** a strongly connected component of G is a maximal strongly connected subgraph.

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.
- 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 computation 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(n³)
  - DFS from every u, v: O(ne) = O(n³)
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)

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

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 1st component)

- All exits \( x \) from \( v \) have \( \#x < \#v \)
- Sufficient to find any of them, e.g. \( \min \# \)

Defn: \( \text{LOW}(v) = \min(\{ \#x \mid x \text{ an exit from } v \} \cup \{ \#v \}) \)

- Calculate inductively:
  \( \text{LOW}(v) = \min \) of:
  - \( \#v \)
  - \( \{ \text{LOW}(w) \mid w \text{ a child of } v \} \)
  - \( \{ \#x \mid (v,x) \text{ is a back- or cross-edge} \} \)

- 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
  - ...

Lemma 3' (in v's SCC)

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 \( r \))
- Contradiction

**SCC Algorithm**

\[
\text{SCC}(v) \quad \text{iv} = \text{DFS number} \\
\quad v.\text{low} = \text{LOW}(v) \\
\quad v.\text{scc} = \text{component} \# \\
\]

\[
\text{iv} = \text{vertex_number}++; v.\text{low} = \text{iv}; \text{push}(v) \\
\text{for}\ \text{all}\ \text{edges}\ (v,w) \\
\quad \text{if}\ \text{iv} == 0 \text{ then} \\
\quad \quad \text{SCC}(w); v.\text{low} = \min(v.\text{low}, w.\text{low}) // \text{tree edge} \\
\quad \quad \text{else}\ \text{if}\ \text{iv} = \text{iv} \&\& w.\text{scc} == 0 \text{ then} \\
\quad \quad \quad v.\text{low} = \min(v.\text{low}, \text{iv}) // \text{cross- or back-edge} \\
\quad \quad \text{if}\ \text{iv} == v.\text{low} \text{ then} \\
\quad \quad \quad \text{scc}++; \\
\quad \quad \text{repeat} \\
\quad \quad \quad w = \text{pop}(); w.\text{scc} = \text{scc}++; // \text{mark SCC members} \\
\text{until\ } w == v \\
\]

**How to Find Exits (in \( k \)th component)**

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

**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”:
  \[
  \text{mark all vertices unvisited} \\
  \text{while there is unvisited vertex} \ v \ \text{do} \\
  \quad \text{soc}(v) \\
  \]
- Exercise: redo example starting from another vertex, e.g. \#11 or \#13 (which become \#1)