CSE 421
Introduction to Algorithms

Depth First Search and
Strongly Connected Components

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

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);

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

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
  — all unvisited vertices reachable from v
  — only unvisited vertices reachable from v
All become descendants of v in the tree.
Proof:
  — dfs follows all direct out-edges
  — call dfs recursively at each
  — 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 need to find loops, which are SCC’s in the program flow graph.
- Nontrivial SCC’s in call-graph are sets of mutually recursive procedures
- If (u, v) means process u is waiting for process v, SCC’s show deadlocks.

Two Simple SCC Algorithms

- u, v in same SCC iff there are paths u → v & v → u
- Transitive closure: $O(n^2)$
- 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 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 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

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 → 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
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 ≤ #x (z ancestor of x; Lemma 2)
– #x < #r (x is an exit from r)
– #z < #r, no z → r path, so return from z first
– Contradiction

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 }

if r is the first root from which dfs returns, then r has no exit

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 → r.
– x is an exit

Cor: If v has no exit, then v is a root.

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 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 ≤ #x (z ancestor of x; Lemma 2)
– #x < #r (x is an exit from r)
– #z < #r, no z → 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)

1st root: LOW(v)=v
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
- \{ LOW(w) | w a child of v \}
- \{ #x | (v,x) is a back- or cross-edge \}

SCC Algorithm

- #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
    - scc#++; repeat
- w = pop(); w.scc = scc#; // mark SCC members
- until w==v

Complexity

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