Algorithms

G. Miller
V. Adamchik

CS 15-451 Spring 2014

Carnegie Mellon University

Graphs - II

Plan:

Strongly Connected Components
Tarjan's Algorithm (1972)

Algorithm for Biconnected Components

Maintain dfs and low numbers for each vertex.

The edges of an undirected graph are placed on a stack as they are traversed.

When an articulation point is discovered, the corresponding edges are on a top of the stack.

Therefore, we can output all biconnected components during a single DFS run.

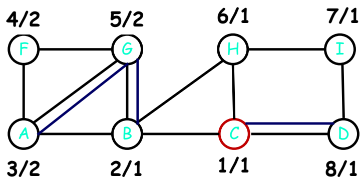
Algorithm for Biconnected Components

```
for all v in V do dfs[v] = 0;
for all v in V do if dfs[v]==0 BCC(v);
k = 0; S – empty stack;
BCC(v)  {
    k++; dfs[v] = k; low[v]=k;
    for all w in adj(v)  do
          if dfs[w]==0 then
                  push((v,w), S); BCC (w);
                  low[v] = min( low[v], low[w] );
                  if low[w] ≥ dfs[v] then  pop(S) ;// output
          else if dfs[w]  < dfs[v]  && w ∈ S then
                  push((v,w), S); low[v] = min( low[v], dfs[w] );
}
```

Algorithm for Biconnected Components

Store edges on a stack as you run DFS

Vertex labels dfs/low



B is an articulation point

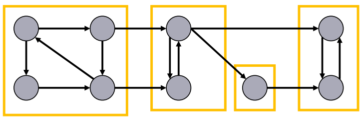
DFS on Directed Graphs

Strongly connected vs. weakly connected

Strongly Connected Components

G is <u>strongly connected</u> if every pair (u, v) of vertices is reachable from one another.

A <u>strongly connected component</u> (SCC) of G is a maximal set of vertices C ⊆ V such that for all vertices in C are reachable.



Equivalent classes
partitioning of the vertices

Two vertices v and w are equivalent, denoted u≡v, if there is a path from u to v and one from v to u.

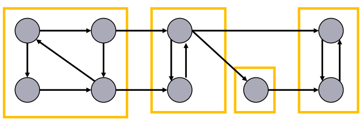
The relation ≡ is an equivalence relation.

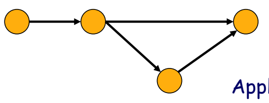
Reflexivity v ≡ v. A path of zero length exists.
Symmetry if v ≡ u then u ≡ v. By definition.
Transitivity if v ≡ u and u ≡ w then v ≡ w
Join two paths to get one from v to w.

The equivalent class of ≡ is called a <u>strongly connected component.</u>

DAG of SCCs

Choose one vertex per equivalent class.
Two vertices are connected if the corresponding components are connected by an edge.





The resulting graph is a DAG.

Applications… social networks

Preamble

<u>Def.</u> low[v] is the smallest dfs-number of a vertex reachable by a back or cross edge from the subtree of v.

<u>Def.</u> A vertex is called a base if it has the lowest dfs number in the SCC.

<u>Lemma 1.</u> Let b be a base in a component X, then any v∈X is a descendant of b and all they are on the path b-v.

<u>Lemma 2.</u> A vertex is a base iff dfs[v] = low[v].

Preamble

WLOG, we assume that there is a vertex in the graph from which there are edges to each other vertex.

If we start a DFS from that vertex, we will get only one spanning tree.

If there is no such a vertex we can always add one. This won't change the other SCCs.

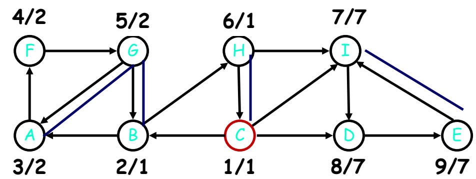
The Algorithm

```
for all v in V do dfs[v] = 0;
for all v in V do if dfs[v]==0 SCC(v);
k = 0; S – empty stack;
SCC(v)  {
    k++; dfs[v] = k; low[v] = k; push(v, S);
    for all w in adj(v)  do
         if dfs[w]==0 then
                  SCC (w); low[v] = min( low[v], low[w] );
         else if dfs[w]  < dfs[v]  && w ∈ S then
                  low[v] = min( low[v], dfs[w] );
   if low[v]==dfs[v] then  //base vertex of a component
         pop(S) where dfs(u) ≥ dfs(v); // output
}
```

The Algorithm

Store vertices on a stack as you run DFS

Vertex labels dfs/low



C on stack

C: low = dfs
pop dfs(v)≥1

I: low = dfs
pop dfs(v)≥7

Correctness

Theorem. After the call to SCC(v) is complete it is a case that

(1) low[v] has been correctly computed

(2) all SCCs contained in the subtree rooted at v have been output.

Proof by induction on calls.

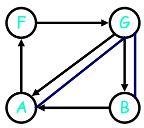
First we prove 1) and then 2).

(1) low[v] correctly computed

```
for all w in adj(v)  do
        if dfs[w]==0 then
                SCC (w); low[v] = min( low[v], low[w] );
        else if dfs[w]  < dfs[v]  && w ∈ S then
                low[v] = min( low[v], dfs[w] );
```

Case a) $w \in S$. Then there is a path w-v. Combining this path with edge (v,w) assures that v and w in the same component.

Case b) $w \notin S$. Then the rec. call to w must have been completed.



(2) all SCCs contained in the subtree rooted at v have been output.

```
if low[v]==dfs[v] then  //base vertex of a component
        pop(S) where dfs(u) ≥ dfs(v); // output
```

By lemma 2, v is a base vertex.

We have to make sure that we pop only vertices from the same component.

Let be another base vertex b that descends from v.

Let assume that there is w (in the same component as v) that descends from both v and b.

There must be a path w-v.

By lemma 1 there is a path v-b. And also b-w.

Cycle w–v–b-w. So, v and b are in the same component.

Lemma 1. Let b be a base in a component X, then any $v \in X$ is a descendant of b and all they are on the path b-v.

Proof. We know that either

(1) v descends from b, or

(2) b descends from v, or

(3) neither of the above.

(2) is impossible since b has the lowest dfs-num.

Suppose (3). There is a path b-v (same component)

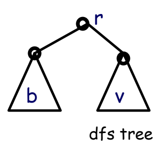
Find the least common ancestor r of all vertices on b-v path. We claim path goes through r.

If so, then dfs[r] < dfs[b]. But r and b are in the same component. (3) is impossible.

Find the least common ancestor r of all vertices on b-v path. We claim path goes through r.

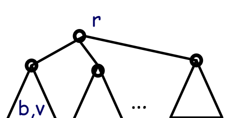
Case 1.

Since dfs[b]<dfs[v], T_b and T_v are disjoint - there are cannot be an edge between them.



Case 2. b and v in the same DFS tree.

b-v path must touch at least two DFS trees, (r is the least)



It follows, b-v path starts in one tree, goes through one or more another subtrees and come back.

Impossible to come back, since dfs-num in one tree is less than in another.