CSE 417: Algorithms and Computational Complexity

Winter 2001
Lecture 10
Instructor: Paul Beame

DFS(v) for a directed graph

DFS(v)

Strongly-connected components

Strongly-connected components

Uses for SCC’s

- Optimizing compilers need to find loops, which are SCC’s in the program flow graph.
- If (u,v) means process u is waiting for process v, SCC’s show deadlocks.
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 \( u, v \): \( O(nm) = O(n^2) \)

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.

Subgoal

- All members of an SCC are descendants of its root.
- 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

A root node \( v \) sometimes has exits 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.

Minimum Spanning Trees (Forests)

- Given an undirected graph \( G=(V,E) \) with each edge \( e \) having a weight \( w(e) \)
- Find a subgraph \( T \) of \( G \) of minimum total weight s.t. every pair of vertices connected in \( G \) are also connected in \( T \)
  - if \( G \) is connected then \( T \) is a tree otherwise it is a forest

First Greedy Algorithm

- Prim’s Algorithm:
  1. Start at a vertex \( v \)
  2. Add the cheapest edge adjacent to \( v \)
  3. Repeatedly add the cheapest edge that joins the vertices explored so far to the rest of the graph.
- We’ll show it works later

Prim’s Algorithm
Second Greedy Algorithm

- Kruskal’s Algorithm
  - Start with the vertices and no edges
  - Repeatedly add the cheapest edge that joins two different components. i.e. that doesn’t create a cycle

- Again we save the proof of correctness for later
Kruskal’s Algorithm produces same tree as Prim’s algorithm.