Data Structures and Algorithms

Spring 2009-2010

Outline

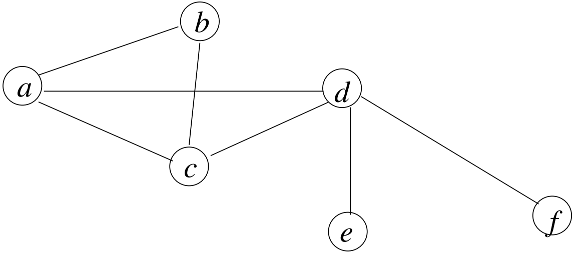
1. Graph Algorithms
   - Biconnectivity
   - Bi-connected Components

Outline

1. Graph Algorithms
   - Biconnectivity
   - Bi-connected Components

Introduction

- A graph is k-connected if there are k vertex-disjoint paths between any pair of vertices
- Thus, a graph is *biconnected* if the removal of any *single* vertex maintains the connectedness of the graph
- If a graph is not biconnected the node(s) that prevent biconnectedness are called *articulation points* or *cut vertices*

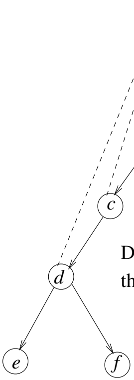


Graph, G

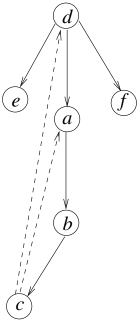
- Graph has one articulation point, d

Introduction (contd.)

- Performing depth-first search on G:



DFS tree for graph, G
that starts at vertex a



DFS tree for graph, G
that starts at vertex d

- The exact shape of the **DFS tree** depends on where we begin our search from

Biconnectivity Algorithm

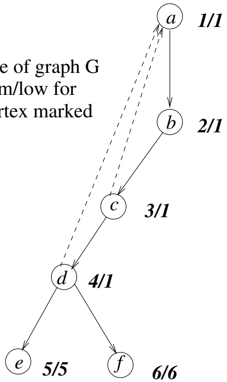
- Starting at vertex *a* its DFS (spanning) tree is shown previously
- Can find all articulation points in a graph G in $O(|V| + |E|)$ (**linear**) time using DFS, and thus tell if a graph is biconnected
- Algorithm:
  - Starting at any vertex, perform a DFS and number the vertices as they are visited
  - Call this vertex number, $num(v)$
  - For each vertex v in the DFS spanning tree, compute $low(v)$, the *lowest-numbered* vertex that is reachable by taking 0 or more forward edges of the tree followed by *at most* one back edge
  - A vertex v is an articulation point iff
    - v is the root of the tree and it has more than one child, or
    - v is any other vertex and it has some child w such that $low(w) \geq num(v)$

Biconnectivity Algorithm (contd.)

Computing *low*(v)

- *num*(v) is computed by numbering the vertices as they appear in a DFS (preorder)
- To compute *low*(v) we can take 0 or more tree edges and 0 or 1 back edges
- *low*(v) is therefore the minimum of
  1. *num*(v) (Rule 1)
  2. the lowest *num*(w) among all back edges (Rule 2)
  3. the lowest *low*(w) among all tree edges (Rule 3)

DFS tree of graph G with num/low for each vertex marked



Biconnectivity Algorithm (contd.)

- How many tree (forward) edges should we take (if any) before we take the back edge (if any)?
- If a node's **descendant** in the tree has a back edge to earlier in the tree then we want to use this back edge
- Can determine this using a *postorder* traversal of the tree
- Can combine preorder *num* and postorder *low* traversals in one call
- Why does algorithm work?
- †Because for a node, v, not to be an articulation point (AP) there must be a back edge from a node later, w, in the tree to one earlier, u – otherwise removing v would make u and w unreachable from each other
- $low(v)$ indicates whether an alternate path exists or not

Correctness of Biconnectivity Algorithm

Theorem

A vertex v is an articulation point iff

1. *v is the root of the tree and it has more than one child*
2. *v is any other vertex and it has some child w such that* $low(w) \geq num(v)$,

Proof

There are two parts to consider.

1. "If v is root of DFS tree then v is AP $\Leftrightarrow$ v has more than one child": Easy (see DFS tree rooted at d in earlier picture).
2. for any other vertex v in DFS tree, v is an AP $\Leftrightarrow$ v has some child w such that $low(w) \geq num(v)$
   $\Leftarrow$: "If v has some child w such that $low(w) \geq num(v)$ then v is an AP"

Correctness of Biconnectivity Algorithm (contd.)

$\Rightarrow$:

- If v is an AP then there must be two vertices, u and w such that the only path from u to w goes through v.
- Assume implication is false. That is, assume theorem reads "If v is an AP then for *all* its children, w, $low(w) < num(v)$."
- Now consider the first AP (lowest $num()$) found in the DFS tree. Then by assumption, any child of v can always "sidestep" v using the backedge implied by $low(w)$ and therefore, v cannot be an articulation point.
- Contradiction.

Outline

The Components of a Graph

- Articulation points separate a graph into **bi-connected components** (BC)
- Vertices u and v are in the same bi-connected component if there are two or more vertex disjoint paths from u to v
- The following code records the bi-connected component of each **edge** since some vertices (namely APs) are members of more than one BC
- The APs can be detected afterwards by checking for vertices that have edges in more than one camp