Depth First Search
Version of Oct 6, 2016

Preamble. In addition to Breadth First Search (BFS), Depth first search (DFS) is another common graph search strategy. It turns out that this search strategy works well with some recursive algorithms for some graph problems. As an example, we will use it solve the articulation point problem.

Algorithm. Contrary to BFS, DFS recursively traces a path as far as possible before returning from the recursion to explore other unvisited neighbors. For solving the articulation point problem, we keep a global variable time and a variable $d[v]$ for each vertex v to record the time at which we first visit v. Note that this is a simplified version of the DFS algorithm we gave in class (since we are only calculating starting times and not finishing times).

DFS(G)

1. time=0;
2. For each vertex u, pred$[u] = -1$ and flag$[u]$:= false.
3. For each vertex u, if flag$[u]$ = false, then DFSVisit(u).

DFSVisit(u)

1. flag$[u]$:= true.
2. time = time + 1 and $d[u]$:= time.
3. For each neighbor v of u, if (flag$[v]$ = false), then pred$[v] := u$ and DFSVisit(v).

If the graph G is connected, every vertex is marked after the vertex u is visited in the for-loop in step 3.

Running time. Let T_u denote the time that we spend at a vertex u, *excluding* the waiting time for any recursive call at the neighbors of u. Then, the running time is

$$O(n) + \sum_u T_u.$$

We have

$$\begin{aligned} \sum_u T_u &\leq \sum_u O(\text{outdeg}(v) + 1) \\ &= O(n + m). \end{aligned}$$

Hence, DFS takes $O(n + m)$ time.
Recall that m is the number of edges and n the number of vertices.

DFS tree. As in the case of BFS tree, the paths traversed by DFS form a DFS tree rooted at the source vertex. We call the edges in a DFS tree the *tree edges*. The edges of G not in the DFS tree are called *back edges*. Recall that if (u, w) is a back edge from u, then w must be on the path from u to the root.

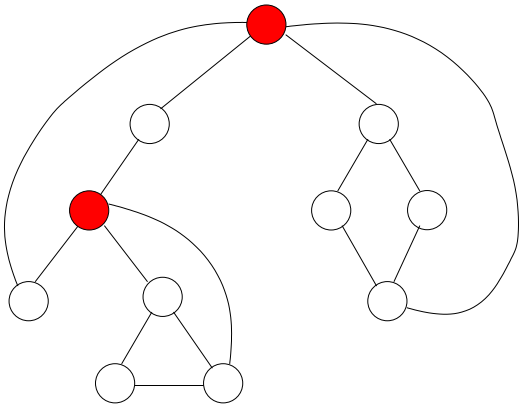
Even if we fix the source vertex, a DFS tree is not unique because its structure depends on the order in which we recursively visit the neighbors of vertices. A useful property is that the start time $d[v]$ increases strictly down a path in a DFS tree.

In particular, note that this implies that if u is a descendent of v and (u, w) is a back edge, then if $d[w] > d[v]$, w is on the path between u and v and if $d[w] < d[v]$, w is on the path between v and the root.

Articulation point. An *articulation point* of G is a vertex whose removal disconnects G. The red vertices in the example below are articulation points. How can we find articulation points quickly?

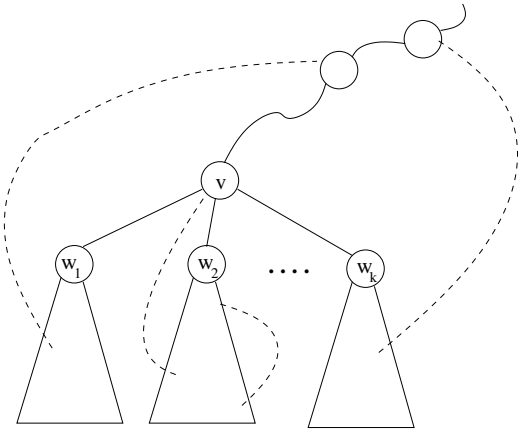
A brute-force approach is to remove each vertex from G and test run a graph search to check the connectivity of the remaining subgraph. This takes $O(n^2 + nm)$ time. We can do better by exploiting the following properties of a DFS tree.

- A leaf of a DFS tree is not an articulation point.
- The root is an articulation point iff it has two or more children.



- An internal vertex v in a DFS tree is an articulation point iff there exists a subtree T rooted at a child of v such that no back edge connects a node in T to a proper ancestor of v.

  Note that if u is a descendent of T, then u has a back edge (u, w) connecting to a proper ancestor of v iff $d[w] < d[v]$.



This means that internal vertex v is an articulation point iff it has a subtree T rooted at a child of v such that no back edge connects T to a vertex w such that $d[w] < d[v]$. This motivates the following definition.

$$low(v) = \min \begin{cases} d(v) & \\ d(w) & (v,w) \text{ a back edge} \\ d(w) & (u,w) \text{ a back edge,} \\ & u \text{ a proper descendant of } v. \end{cases}$$

which can be written recursively as

$$low(v) = \min \begin{cases} d(v) & \\ d(w) & (v,w) \text{ a back edge} \\ low(u) & u \text{ a child of } v \end{cases}$$

The recursive formulation can be translated into a DFS-like algorithm to compute low[v] for each vertex v. Just modify DFS so that, when v is being processed for the first time it checks every one of its neighbors w to see if (v, w) is a back edge or a child edge (or if w is v's parent). It starts by setting $low[v] = v$. If the current edge being checked is a back edge it sets $low[v]$ to be the minimum of $low[v]$ and $d[w]$. If it's a child edge it recursively calculates $low[w]$ and sets $low[v]$ to be the minimum of $low[v]$ and $low[w]$. Note that the recursion stops when v is a leaf in the tree, i.e., when v has no children (recall that in this case, v is NOT an articulation point). $low[v]$ can then be used to check if v is an articulation point.

The full pseudocode is below. It starts with all nodes v having $flag[v] =$ false and $pred[v] = -1$.

Art(v)

1. flag[v] := true, time := time + 1, $d[v]$:= time, low[v] := $d[v]$.
2. For each neighbor w of v,
   (a) If flag[w] = false, then
      i. pred[w] := v.
      ii. Art(w).
      iii. If pred[v] = -1 and w is v's second child, output v.
      iv. Otherwise, if low[w] $\geq d[v]$, output v.
      v. low[v] := min(low[v], low[w]).
   (b) Otherwise, if $w \neq$ pred[v], then low[v] := min(low[v], $d[w]$).

The overall running time is still $O(n + m)$ because only $O(1)$ extra instructions are added into the DFS pseudocde and each takes $O(1)$ time.

Note: This document was written by M J. Golin, revised from an original by S.W. Cheng, for COMP3711H, HKUST.